Why Caret – Simplifies accessing different prediction packages.

PML_2-1Caret Package Caret Package url: http://topepo.github.io/caret/index.html

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

This lecture's about the caret package, which is a very useful front end package that wraps around a lot of the prediction algorithms and tools that you'll be using in the R programming language. Find the caret package at the website that I linked to or you can just Google caret R package. The functionality that's built into the caret package includes some of the following. For example, we can use the preprocessing tools in the caret package to clean data and get the features set up, so that they can be used for prediction. We can also do cross validation and data splitting within the training set, using the create DataPartition and create TimeSlices functions. We can also create training and test sets with the training and predict functions. And we can use those to train data sets at train prediction functions and apply them to new data sets. We can also do model comparison using the confusion matrix function, which will give you information about how well the models did on new data sets.

There a large machine learning algorithms that are built into R, so these range from very popular statistical machine learning algorithms like ***linear discriminant analysis*** and regression to much more widely used algorithms in computer science, like ***support vector machines***, ***classification and regression trees***, ***random forests*** or ***boosting***. All of these algorithms are built by a variety of different developers with different backgrounds results in slightly different interfaces for each of these prediction algorithms. One example, consider this class of different prediction algorithms that you could have applied, everything from linear discriminate analysis down to boosting. For each of these different algorithms, you can imagine creating an object called **objnr**. That object will have a different class, say a ***linear discriminate analysis*** or **glm** and so forth. And for each of these objects if we apply the predict function, we have to pass slightly different parameters each time in order to get the prediction of the outcome. For example, from the **glm** package, we have to say **`type =" response"`** to get the prediction of the response from that model fit. Or, for example, if we want to use **rpart**, we want to predict with **`type=" probability"`** in order to predict the response. In each case, they're a little bit different, and *the caret package provides a unifying framework that allows you to predict using just one function and without having to specify all the options that you might care about in order to get the same prediction out.*

Spam Example:

So here's a quick example, using the caret package. We'll go into the details of how this is done very specifically in later examples. So here we've loaded in the caret package, and we've loaded the kern lab package as well to get the spam data set. And so, what we can do first is partition the data setup into a training and a test set. Here I'm going to use the spam type, and I'm going to split it into the, training set and the test set. I'm going to say we're going to use about 75% of our data to train the model and 25% to test. Then what I can do is I can actually subset the data into the training data using the in train, bit, the

in train, object that comes out from create data partition, and I can create the testing data set by finding all those samples that aren't in the training set. Then this will give me a subset of that data that are just for training and a subset of the data that adjust for testing. And you can do this with sort of a simple interface.

```
library(caret); library(kernlab); data(spam)
inTrain <- createDataPartition(y=spam$type,
                               p=0.75, list=FALSE)
training <- spam[inTrain,]
testing <- spam[-inTrain,]   # -inTrain is  all data not in inTrain set
dim(training)
```

```
[1] 3451   58
```

Next you can fit a model, so here I'm going to use the train command from the caret package, and so again I'm trying to predict type. And I use the tilde and the dot to say use all the other variables in this data frame in order to predict the type. And I tell which data set I want to build the training model on, and so, in this case, the training data set we created on the previous slide. And then I just tell which of the methods that I'd like to use, and so you can use GLM or you can use a bunch of other different models. And so what this does is it'll create a model fit from the train function, and as we use the 3451 samples in a training set and the 57 predictors to predict which class you're belonging to based on a model, a GLM model. And so what it can do is it can do a bunch of different ways of testing whether this model will work well and using it to select the best model. And in this case it used resampling. And it does bootstrapping with 25 replicates, and it corrects for the potential bias that might come from bootstrap sampling.

```
set.seed(32343)
modelFit <- train(type ~.,data=training, method="glm")  # train from caret ,  ~.  means use all
variables in data set for prediction
modelFit
```

```
Generalized Linear Model

3451 samples
  57 predictors
   2 classes: 'nonspam', 'spam'

No pre-processing
Resampling: Bootstrapped (25 reps)

Summary of sample sizes: 3451, 3451, 3451, 3451, 3451, 3451, ...

Resampling results

  Accuracy  Kappa  Accuracy SD  Kappa SD
  0.9       0.8    0.02         0.04
```

So once we fit that model, we can actually look at the model, and so the way we can do that is look at the finalModel component of the modelFit object from **`modelFit$finalModel`**. It will tell you what the actual fitted values are from the GLM model.

```
modelFit <- train(type ~.,data=training, method="glm")
modelFit$finalModel  # modelFit $finalModel gives the final model coefficients.
```

```
Call:  NULL
```

```
Coefficients:
      (Intercept)              make           address               all             num3d
        -1.78e+00        -7.76e-01        -1.39e-01         3.68e-02         1.94e+00
              our             over            remove          internet            order
         7.61e-01         6.66e-01         2.34e+00         5.94e-01         4.10e-01
              mail          receive             will           people           report
         4.08e-02         2.71e-01        -1.08e-01        -2.28e-01        -1.14e-01
         addresses             free         business            email              you
         2.16e+00         8.78e-01         6.49e-01         1.38e-01         6.91e-02
            credit             your             font          num000            money
         8.00e-01         2.17e-01         2.17e-01         2.04e+00         1.95e+00
                hp              hpl           george          num650              lab
        -1.82e+00        -9.17e-01        -7.50e+00         3.33e-01        -1.89e+00
```

Then you predict on new samples by using the **predict** command. Again, it's a unified framework, so we just type predict. We pass it the modelFit that we got from the train, function in carrot, and we pass it which data we would like it to predict on. So in this case, the new data is the testing data. When you do that, it will give you a set of predictions that correspond to the responses, and you can use those to try to evaluate whether your model fit works very well or not.

```
predictions <- predict(modelFit,newdata=testing)  # predict using modelFit parameters applied to
testing data
predictions
```

```
  [1] spam    spam    spam    nonspam nonspam nonspam spam    spam    spam    spam    spam
 [12] spam    spam    spam    spam    spam    spam    spam    nonspam spam    spam    spam
 [23] nonspam spam    nonspam nonspam spam    spam    spam    spam    spam    spam    spam
 [34] spam    spam    spam    spam    spam    spam    spam    spam    spam    spam    spam
 [45] spam    spam    spam    spam    nonspam spam    nonspam spam    spam    spam    spam
 [56] spam    nonspam nonspam spam    spam    spam    spam    spam    nonspam spam    spam
 [67] spam    spam    spam    spam    spam    spam    spam    spam    spam    spam    spam
 [78] nonspam nonspam nonspam spam    spam    nonspam spam    nonspam nonspam spam    spam
 [89] spam    spam    spam    spam    spam    spam    nonspam spam    spam    spam    spam
[100] spam    spam    spam    nonspam spam    nonspam spam    spam    spam    spam    spam
[111] spam    spam    spam    spam    nonspam spam    spam    spam    spam    spam    spam
[122] spam    spam    spam    spam    spam    spam    spam    nonspam spam    spam    nonspam
[133] spam    spam    spam    spam    spam    spam    spam    spam    spam    spam    spam
[144] spam    spam    spam    nonspam spam    spam    spam    spam    spam    spam    spam
[155] nonspam spam    nonspam spam    nonspam spam    spam    spam    spam    spam    spam
[166] spam    spam    spam    spam    spam    spam    spam    spam    spam    spam    spam
```

Confusion Matrix and Statistics

One way that you can do that is by calculating the confusion matrix, so that's using this confusion matrix function. Note the capital M. Don't miss that when you're typing **confusionMatrix**. Then pass in the predictions that you got from your model fit; then the actual outcome on the testing samples. In this case, it was the type - whether it was spam or ham message. And then it will record the confusion matrix. It will generate a table for which of the cases that you predicted to be nonspam or actually nonspam, which is the cases where it was spam, and you predicted to be spam and so forth. Then follows a bunch of summary statistics. For example, the accuracy, a 95 percent confidence interval for the accuracy, and then a bunch of information about how well they correspond in other categories. The sensitivity and the specificity of that. The confusion matrix function wraps a bunch of different accuracy measures that you might want when you're evaluating the model Fit.

```
confusionMatrix(predictions,testing$type)  # evaluate predictions, recall we are testing type, spam or nonspam
```

```
                    Reference
Prediction nonspam spam
   nonspam     665     54
          spam          32  399
              Accuracy :       0.925
                95% CI : (0.908, 0.94)
    No Information Rate : 0.606
    P-Value [Acc > NIR] : <2e-16

                 Kappa : 0.842
 Mcnemar's Test P-Value : 0.0235
            Sensitivity : 0.954        #   P(declare spam | spam) ??
    Specificity : 0.881    # P(declare nonspam | nonspam) ??
```

Caret tutorials:

These tutorials are good and are very useful for covering material that we don't cover in this class. There's also a good paper in the journal of statistical software that introduces the caret package if you want further information.

http://www.edii.uclm.es/~useR-2013/Tutorials/kuhn/user_caret_2up.pdf
http://cran.r-project.org/web/packages/caret/vignettes/caret.pdf
A paper introducing the caret package
http://www.jstatsoft.org/v28/i05/paper

PML_2-2 Data Spliting / Slicing

So this lecture is about data slicing, and you might use data slicing either for building your training and testing sets right at the beginning of your prediction function creation, or you might use it for performing cross validation or boot strapping within your training set, in order to evaluate your models.

Data Spliting Example

I've loaded the caret package and the kernlab data set again so I can load the spam data. This is a data set where we're trying to predict whether emails are spam or ham and we have a bunch of variables that measure how many times a particular word appears, or how often capital letters appear. We can use createDataPartition, to separate the data set into training and test sets. inTrain is a list (TRUE/FALSE) of which outcome I want to split on, so in this case on the "type". I want to create a data set that's 75% is allocated to the training set and 25% is allocated to the testing set. The inTrain variable gets assigned an indicator function that I can use to subset out the training set and the test set. The training set created by subsampling this spam data set, the spam data frame, into only those samples that appear in the training set. And then, the testing set is all the remaining samples, by using this minus sign to indicate that they're, all the samples that don't appear in this index set in the inTrain variable. This is one way to split your data into a training and a test set right at the beginning.

```
library(caret); library(kernlab); data(spam)
inTrain <- createDataPartition(y=spam$type,
                  p=0.75, list=FALSE)  # partition on type, use 75% for training, 25% for test, row indexed
training <- spam[inTrain,]
testing <- spam[-inTrain,]  # all data not in inTrain becomes test set
dim(training)
```

```
[1] 3451  58
```

SPAM Example: K-fold

To do cross validation you split your training set into a bunch of smaller data sets, The K-folds that you'll then use to do the cross validation. One way that you can do that is with this **`createFolds`** function in the kay caret package. Again, you pass it the outcome that you want to split on, the **`spam $type`** variable. Then input the number of folds to create, in this case 10 folds. And in this case, set **`list=TRUE`**, which means it will return each set of row indices corresponding to each particular fold as lists (10 here) within a list. Can also either return the training set **`returnTrain=TRUE`** or not return the training set. So, for the example 10 folds have been created.

```
set.seed(32323)
folds <- createFolds(y=spam$type,k=10,
                  list=TRUE,returnTrain=FALSE)  # operating on type, 10 folds,  lists folds, and return training set.
```

And to check the length of each fold use **`sapply(folds, length)`**. For example, in Fold01 there are 4141 samples, in Fold02, there's 4140, and so forth. So it split the data set up into ten folds, where each fold has approximately the same number of samples in it.

```
sapply(folds,length)  # shows lengths in each fold.
```

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

Take the first element of the fold's list to look at which samples appear in the first fold and the first 10 elements in the sample. In other words, this is split up in order into the first to 4100 samples in the first fold. The second 4140 samples in the second

fold and so forth, so you can actually look at which of the indices corresponding to the training set and the test set within those K-folds.

```
folds[[1]][1:10]  # displays first ten items  of first fold
```
```
[1]  1  2  3  4  5  6  7  8  9 10
```

SPAM Example: Return test # return test set (or the training set)

You can have it also return the test set, or you can have it return the training set. So remember I'm, here I'm splitting the data set about 75, 25 into training and test sets and so what I can do is I can say **returnTrain=FALSE**. Then it will actually return just the test set samples.

```
set.seed(32323)
folds <- createFolds(y=spam$type,k=10,
                             list=TRUE,returnTrain=FALSE)  # operating on type, 10
folds,  lists folds, and not return training set., test set only.
```

See that there's much smaller number of samples in each fold, because most of the samples are going to the training set.

```
sapply(folds,length)  # shows lengths in each fold.
```
```
Fold01 Fold02 Fold03 Fold04 Fold05 Fold06 Fold07 Fold08 Fold09 Fold10
   460    461    460    459    461    459    460    460    461    460
```

For **fold 01**, see the samples that actually correspond to the testing set in that fold.

```
folds[[1]][1:10] # displays first ten items  of test set, note 460 ~ 25% of 460 +4601 = 5061
```
```
[1] 24 27 32 40 41 43 55 58 63 68
```

Resampling Example (or bootstrapping)

If instead of doing a full cross validation, you want to do something like resampling or bootstrapping, you can use the **createResample** function, resample function. Specify how many times you would like to resample the data and whether you would like a list out, or vector out, or matrix out.

```
set.seed(32323)
folds <- createResample(y=spam$type,times=10,
                             list=TRUE)  #  number of resamples, list out or matrix out
or vector out; list in this example.
```

```
sapply(folds,length)
```
```
Resample01 Resample02 Resample03 Resample04 Resample05 Resample06 Resample07 Resample08 Resample09
      4601       4601       4601       4601       4601       4601       4601       4601       4601
Resample10
      4601
```

So, for example, in the first fold, you actually get the sample three repeated three different times in that fold because you're re-sampling with replacement from the values.

```
folds[[1]][1:10]
```
```
[1]  1  2  3  3  3  5  5  7  8 12     # sample 3 occurred 3 times in first 10 samples of fold 1
```

Time Slices Example

If you areanalyzing data that you might be using for forecasting, you can use it to check what are the ***time slices*** where you take continuous values in time. Here I've created a time vector that's 1 to 1000. The integer is 1 to 1,000; create slices that have a window of about 20 samples in them; predict the next 10 samples out after taking the initial window of 20.

```
set.seed(32323)
tme <- 1:1000  # where to take continuous values in time
folds <- createTimeSlices(y=tme,initialWindow=20,
                          horizon=10)
# take a window of 20 samples, and predict out 10 samples.
names(folds)
```

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

There are 20 values in the first training set . Twenty (20) samples that have been created in the first window that will to be used to predict.

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

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

The test set has 10 samples, 21 to 30. And the next subsample would be shifting that over and getting a next time slice, but it gets continuous sets of numbers all in a row so that you can use the time varying component of the sample in order to predict. There's a lot of information in the caret tutorials about how to do time slicing, and we'll show various examples of it when we build prediction functions in the rest of this class, and your homework will cover this as well.

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

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

PML_2-3 Training Options

This is a brief lecture about some about the training control options that you have when training models using the Caret package. Will be using the SPAM example just to illustrate how these ideas work.

Spam Example

Load the Caret package, the **kernlab** package and attach the spam data set. Use the **createDataPartition** function to create a set of indices corresponding to the training set; set about 75% of the data to be in the training set. Then define **training** and **testing** sets using those indicator functions. Use the train function and basically set all of the defaults to be whatever defaults that the train function chooses for you; other than maybe the method to fit, **method=**, and which dataset, **data=**.

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

Train Options

You can use a large set of options for training. Here are a few. One, you can use this **preProcess** parameter to set a bunch of pre-processing options. You can also set weights; you can up-weight or down-weight certain observations. It is particularly useful with a very unbalanced training set that has a lot more examples of one type than another. You can set the metric so by default for factor variables. That is for categorical variables, the default metric is accuracy that it's trying to maximize. For continuous variables it's the root mean squared error. You can also set a number of other control parameters using the **trControl** parameter. You have to pass it the call to the **trainControl** function.

```
args(train.default)
function (x, y, method = "rf", preProcess = NULL, ..., weights = NULL,
    metric = ifelse(is.factor(y), "Accuracy", "RMSE"), maximize = ifelse(metric ==
        "RMSE", FALSE, TRUE), trControl = trainControl(), tuneGrid = NULL,
    tuneLength = 3)    # preProcess data, add weights to variables,  "accuracy" for factors,
"RMSE" for continuous
# can use RMSE or Rsquared for continuous; Accuracy (fraction correct) or Kappa (concordance) for
factors
NULL
```

Metric options

Metric options built into the train function for continuous outcomes are **RMSE** or root-mean-squared error and **RSquared**. **RSquared** is the r squared that you get from a regression model, a measure of linear agreement between the variables that you're predicting and the variables that you predict with. Linear agreement is very useful I when using linear regression. It may not be useful for non-linear things, like random force. Accuracy is the fraction correct, the default measure of accuracy for categorical outcomes. Kappa is a more complicated measure that is frequently used in competitions like Kaggle.

Continous outcomes:

- RMSE = Root mean squared error
- RSquared = R2 from regression models

Categorical outcomes:

- Accuracy = Fraction correct
- Kappa = A measure of concordance, http://en.wikipedia.org/wiki/Cohen%27s_kappa

Train Control

The **trainControl** argument allows you to be much more precise about the way that you train models. You can tell it which method to use for resampling the data, whether it's bootstrapping or cross validation. You can tell it the number of times to do bootstrapping or cross validation. You can also tell it how many times to repeat that whole process if you want to be careful about repeated cross validation. You can tell at the size of the training set with the **p =** parameter, and then you can tell it a bunch of other parameters that depend on the specific problems you're working on. For example, for time course data, **initialWindow** tells you the size of the training data set, the size of the number of time points that will be in the training data. And a horizon is the number of time points that you'll be predicting. You could also have it return the actual predictions themselves from each of the iterations when its building the model. You could also have it return a different summary with a different **summary Function** or a default summary if you like it to. And then you can set preprocessing options as well as things like prediction bounds and can set the seeds for all the different resampling layers, this is particularly used if you are going to be parallelizing your computations across multiple cores. We are not going to cover that too extensively here but if you're training models on large numbers of samples with a high number of predictors using parallel processing it can be highly useful for [im]proving the computational efficiency of your analysis.

```
args(trainControl)
function (method = "boot", number = ifelse(method %in% c("cv",
    "repeatedcv"), 10, 25), repeats = ifelse(method %in% c("cv",
    "repeatedcv"), 1, number), p = 0.75, initialWindow = NULL,
    horizon = 1, fixedWindow = TRUE, verboseIter = FALSE, returnData = TRUE,
    returnResamp = "final", savePredictions = FALSE, classProbs = FALSE,
    summaryFunction = defaultSummary, selectionFunction = "best",
    custom = NULL, preProcOptions = list(thresh = 0.95, ICAcomp = 3,
        k = 5), index = NULL, indexOut = NULL, timingSamps = 0,
    predictionBounds = rep(FALSE, 2), seeds = NA, allowParallel = TRUE)
# see reference materials and notes below
NULL
```

Train Control resampling

There are methods that are offered for resampling, again passed the **trainControl** function. You can use standard bootstrapping, you can use bootstrapping that adjusts for the fact that multiple samples are repeatedly resampled when you're doing that sub sampling. This will reduce some of the bias due to the bootstrapping. You can use cross validation (talked about in previous lectures). You could also use repeated cross validation if you want to do sub cross validation with different random draws. You can use "leave one out cross validation" and remember there's a bias variance trade-off between using large number of folds and smaller number of folds when doing cross validation. You can also tell it the number of bootstraps samples or the number of subsamples to take and the number of times to repeat that subsampling if you're doing something like repeated cross validation. In general the defaults work pretty well, but if you have large numbers of samples, or you have a model that requires fine-tuning across a large number of parameters, you may want to increase, for example, the number of cross-validation or bootstrap samples that you take.

- *method*
  - *boot* = bootstrapping
  - *boot632* = bootstrapping with adjustment [for bias]
  - *cv* = cross validation
  - *repeatedcv* = repeated cross validation
  - *LOOCV* = leave one out cross validation
- *number*
  - For boot/cross validation
  - Number of subsamples to take

- *repeats*
  - Number of times to repeat subsampling
  - If big this can *slow things down*

Setting the seed

An important component of training these models is setting the seed. Most of these procedures rely on random resampling of the data. And if you rerun the protocol over again, or rerun the code over again, you will get a slightly different answer because there was a random draw that was created when you were doing cross-validation. If you set a seed, a random number seed, what that will do is that will ensure that the same random numbers get generated each time. The computer is generating pseudo random numbers and if you set the seed it will generate the same sequence of pseudo random numbers again. So you can, it's very useful, often to set the overall seed. This is the seed for the entire procedure so you can get repeatable results. If you're doing parallel computation, you can also set the seed for each resample. You can do that with the seed argument to the **`trainControl`** function. Seeding each resample is particularly useful for parallel fits but is often not necessary when you're doing all your processing that isn't parallel.

- It is often useful to set an overall [random number] seed
- You can also set a seed for each resample
- Seeding each resample is useful for parallel fits

Seed Example

So here's an example of that. If I set the seed using the set.seed function in r, and then I give it a number, an integer number, it will set a, a seed that's consistent for performing this analysis, and so it'll generate a set of random numbers that is consistent. So then if I fit my model like this, then when it generates bootstrap samples,. It will generate those bootstrap samples according to the random numbers that come from this seed, if I then reset the seed again, and fit the model again, now with the different number **`modelFit3`** instead of **`modelFit2`**. Then I will get exactly the same bootstrap samples and exactly the same measures of accuracy back out again. This is important when you're training models and then you want to share your training data set with someone else and ensure that you get the same answer when they run the same code.

```
set.seed(1235)    # makes sure same pseudorandom starting point for repetitions
modelFit2 <- train(type ~.,data=training, method="glm")
modelFit2
```

```
Generalized Linear Model
3451 samples
  57 predictors
   2 classes: 'nonspam', 'spam'
No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 3451, 3451, 3451, 3451, 3451, 3451, ...
Resampling results
  Accuracy  Kappa  Accuracy SD  Kappa SD
  0.9       0.8    0.007        0.01
```

Seed Example (showing fixing seed generates same result)

```
set.seed(1235)
modelFit3 <- train(type ~.,data=training, method="glm")
modelFit3
```

```
Generalized Linear Model
3451 samples
  57 predictors
   2 classes: 'nonspam', 'spam'
No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 3451, 3451, 3451, 3451, 3451, 3451, ...
Resampling results
 Accuracy  Kappa  Accuracy SD  Kappa SD
 0.9       0.8    0.007        0.01
```

Further resources

There is more information about this in the Caret tutorial and also in this document about model training and tuning with the Caret package.

- Caret tutorial
- Model training and tuning

PML_2-4 Plotting Predictors

One of the most important components of building a machine learning algorithm or prediction model is understanding how the data actually look and how the data interact with each other. The best way to do that is actually by plotting the data, in particular plotting the predictors.

Wage Data Data from: ISLR package from the book: Introduction to statistical learning

We're going to be using wages data from the ISLR package and it's from the book Introduction to Statistical Learning. We're using this data and to see how we can use it for prediction. I load the ISLR package (install it first and then load it). Load the ggplot2 package for plotting and the caret package for model building. The wage data is actually in the ISLR package; can load it with **data(Wage)**. Look at a summary of that wage data to look at the different variables in Wage. We have the year of the data that's collected and the age of the person who is the data is collected on. It's only male people in this data set, the marital status of those people, their race, education, the region where they were (Mid Atlantic region), the different kinds of job classes, and their health. This gives you information about the data that we're looking at. We've detected a few interesting characteristics of this data just by looking at a summary. We know that they're all men. We know that they're all in the Mid Atlantic region, for example.

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

```
      year             age                    sex                         maritl
           race
 Min.   :2003   Min.   :18.0   1. Male  :3000   1. Never Married: 648   1.
White:2480
 1st Qu.:2004   1st Qu.:33.8   2. Female:   0   2. Married      :2074   2.
Black: 293
 Median :2006   Median :42.0                    3. Widowed      :  19
       3. Asian: 190
 Mean   :2006   Mean   :42.4                    4. Divorced     : 204
4. Other:  37
 3rd Qu.:2008   3rd Qu.:51.0                    5. Separated    :  55
 Max.   :2009   Max.   :80.0



             education                     region                     jobclass
                health
 1. < HS Grad      :268   2. Middle Atlantic   :3000   1. Industrial :1544   1. <=Good     :
858
 2. HS Grad        :971   1. New England       :   0   2. Information:1456   2.
>=Very Good:2142
 3. Some College   :650   3. East North Central:   0
 4. College Grad   :685   4. West North Central:   0
 5. Advanced Degree:426   5. South Atlantic    :   0
                          6. East South Central:   0
```

Get training / test sets

Build a training set and a test set. Set aside the testing set to look at the data at the end of the model building to apply it just one time. Do all of the plotting in the training set.

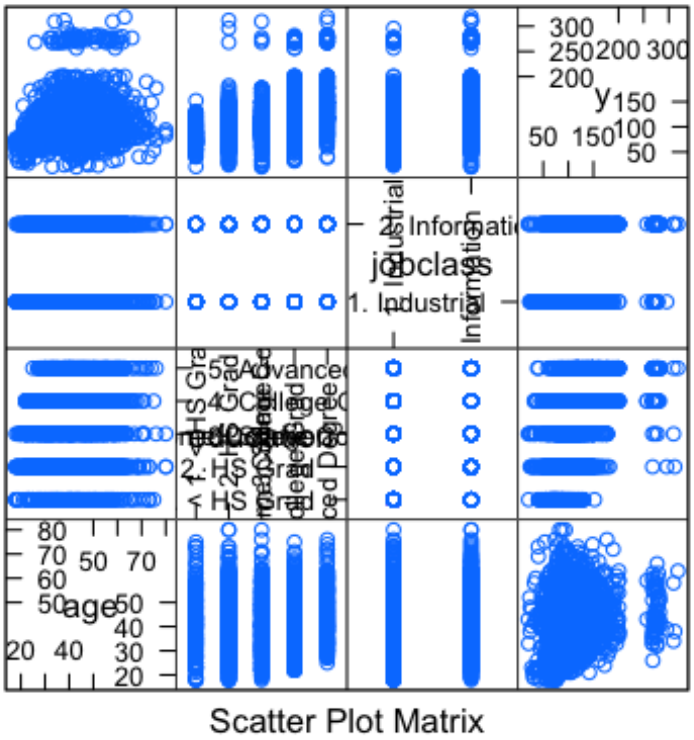
```
inTrain <- createDataPartition(y=Wage$wage,
                               p=0.7, list=FALSE)
training <- Wage[inTrain,]
testing <- Wage[-inTrain,]
dim(training); dim(testing)
```

```
[1] 898  12
```

Feature Plot (caret package)

One example is to use this feature plot from the caret package. This will plot all of the features against each other. Sadly this plot is confusing for this data. The outcome is the wage versus different variables; age, education, and job class. The y variable is the outcome (Wage) that we care about, and here are the different variables. I've got age, education, that are not clear in this plot and job class. The is the plot here of the outcome y versus the job class. You can do that for every box, job class versus education and it's the same plot with the axis reversed. In particular, what you are looking for are all the plots corresponding to each of the variables plotted versus the y variable. Look for any variable that seems to show a relationship with the y variable. For example, there seems to be a trend here between education and salary. So this is one way that you can look at all the data.

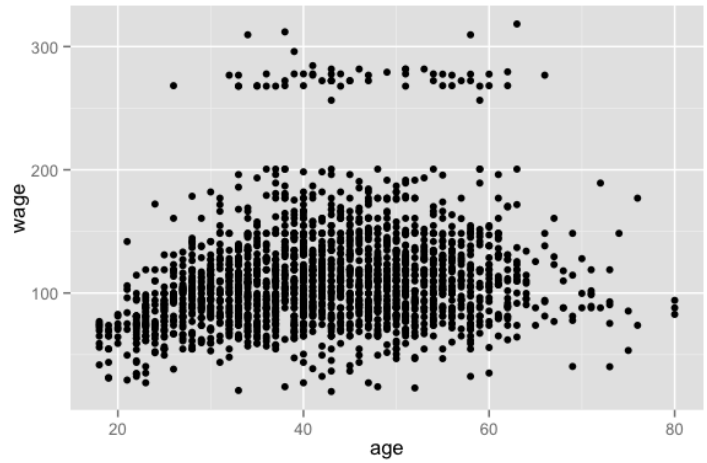
```
featurePlot(x=training[,c("age","education","jobclass")],
            y = training$wage,
            plot="pairs")
```



Qplot (ggplot2 package)

Use either **qplot**, function in the **ggplot2** package or just the plot function base R. So here I'm plotting age versus wage, and so you can see again it appears that there seems to be some kind of trend with age and wages. But you also see, one thing that you notice frequently from making plots like this. Here's some very strange patterns. So you see there's this big chunk up here of observations that appear to be very different than the relationship down here for these chunks. So one thing that we might want to do is try to figure out why there, there's that strange relation between ages and wages before we build our, our model. [Note strange data above major mass of wage-age relationship.]

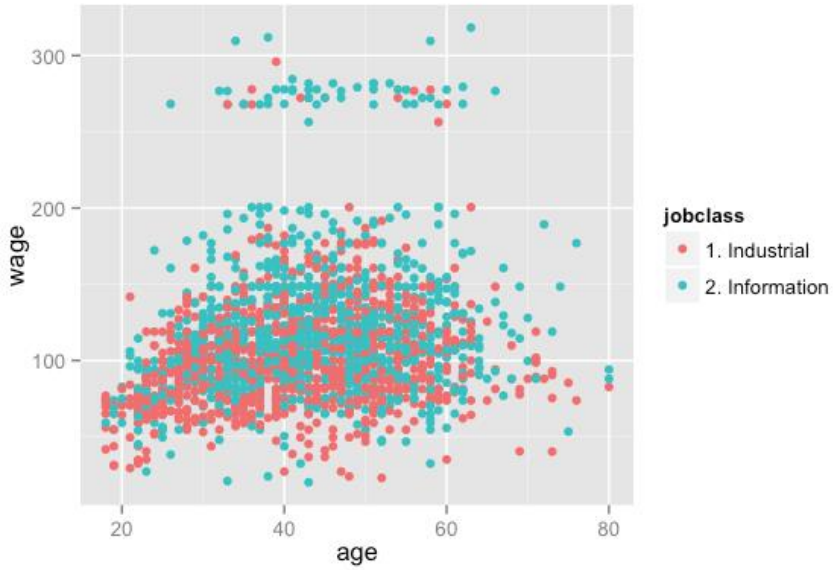
```
qplot(age,wage,data=training)
```



Qplot with color (ggplot2 package)

Use the ggplot2 package to color that plot by different variables, age versus wage. The x axis is age. The y axis is wage. The data points are now colored by the job class by passing the parameter color to the plot function. Note that most individuals that are up in this upper chunk come from the information based jobs as opposed to the industrial jobs. That might explain a lot of the difference between these two big classes of observations. This visually gives a way to detect variables that might be important because they show variation in the data.

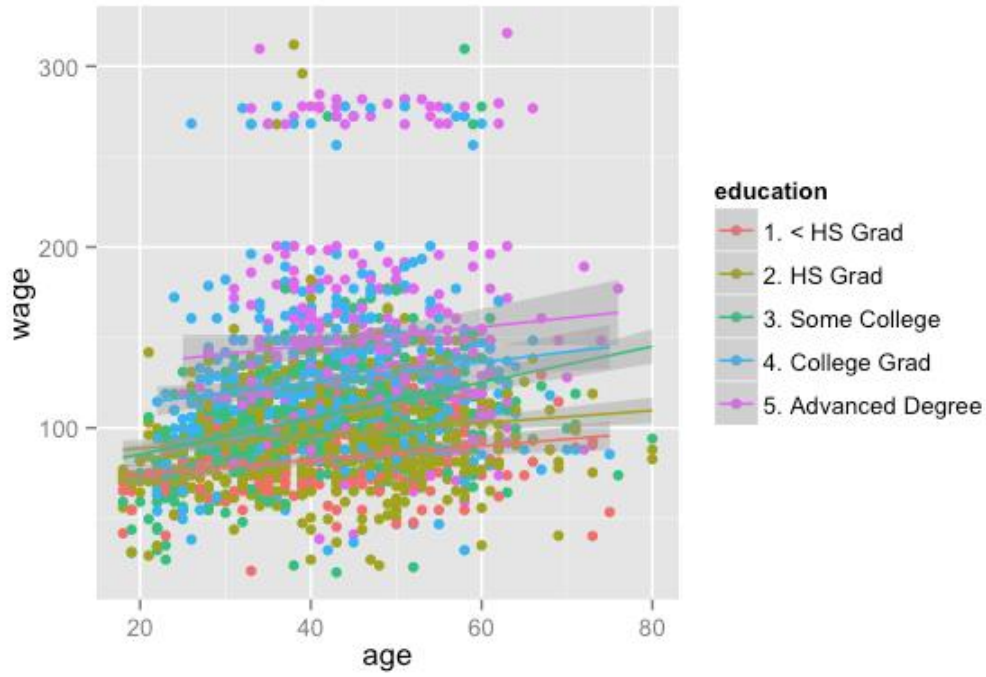
```
qplot(age,wage,colour=jobclass,data=training)
```



Add regression smoothers (ggplot2 package)

You can add regression smoothers as in this plot of age versus wage colored by education. Use the **geom_smooth** function to apply a linear smoother to the data for every different education class, it fits a linear model. The purple line corresponds to people with advanced degrees. The green line corresponds to people with some college. You can see if there's a different relationship for different age groups.

```
qq <- qplot(age,wage,colour=education,data=training)
qq +  geom_smooth(method='lm',formula=y~x)  # apply linear smoother to the data by class
```



Cut2, making factors (*Hmisc* package)

It is often very useful is to break up things like the wage variable into different categories if it's clear that specific categories seem to have different relationships. This can be done with the cut2 function, that's in the Hmisc package. Load the Hmisc package. Then use **cut2** with the g parameter to specify ow many groups to break the data set into. It will break the data set into factors based on quantile groups.

```
cutWage <- cut2(training$wage,g=3)  # cut wages into 3 category levels
table(cutWage)
```

All of the observations that land between 20.1 and 91.7 on the wage variable will get assigned to the first factor level and then all the values between 91.7 and 118.9 will get assigned to the next level and so forth. You can use that to make different plots.

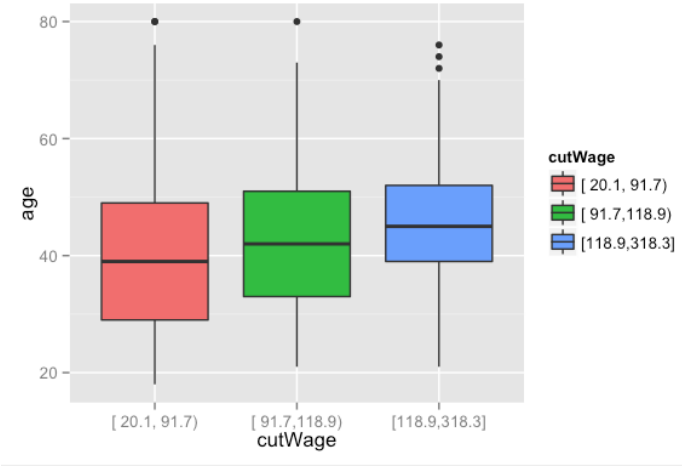
```
cutWage
```

```
[ 20.1, 91.7) [ 91.7,118.9) [118.9,318.3]
          704           725           673
```

Boxplots with cut2

Plot wage groups versus age with **qplot** but now pass it the box plot geometry. You can see here a little bit more clearly the relationship between age and wage.

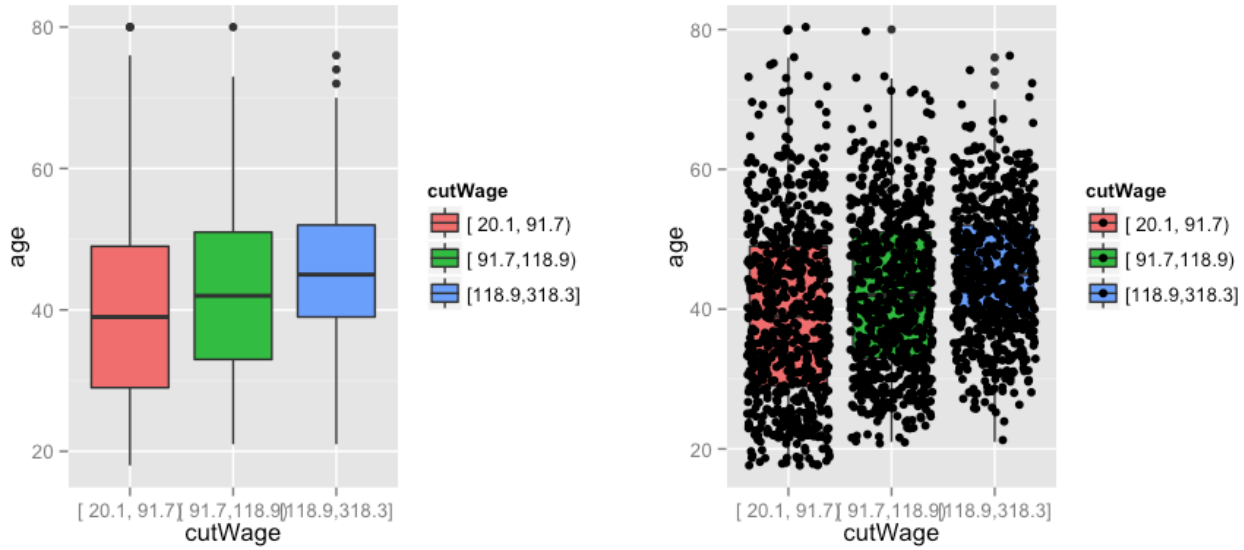
```
p1 <- qplot(cutWage,age, data=training,fill=cutWage,
      geom=c("boxplot"))
p1
```



Boxplots with points overlayed

The points themselves can by overlayed. Sometimes box plots can obscure how many points are being shown and so pass it both box plot and jitter and arrange the plot to see both the box plot itself and the box plot with points overlaid with grid arrange that generates the two plots. p1 was the plot made on the previous slide and p2 is the plot made with points overlaid. And **grid.arrange** makes two plots side by side. There's a large number of dots in each of the different boxes that suggest that this trend may actually be real. If there are just a few dots in the boxes it means maybe that that particular box isn't very well representative of what the data actually looks like.

```
p2 <- qplot(cutWage,age, data=training,fill=cutWage,
      geom=c("boxplot","jitter"))  #
grid.arrange(p1,p2,ncol=2)    # creates the two plots, note p1 and p2
```



Tables

The cut variable, the factorized version of the continuous variable, can be used to look at tables of data. Make a table comparing the factor version of wages to the job class. There are more industrial jobs in the lower wage variable than there are information jobs. The trend reverses for the higher wage jobs with fewer industrial and more information people.

```
t1 <- table(cutWage,training$jobclass)  # cut Wage by training job class
t1
```

```
cutWage         1. Industrial 2. Information
  [ 20.1, 91.7)           437            267
  [ 91.7,118.9)           365            360
  [118.9,318.3]           263            410
```

Get the proportions in each group by passing it two to give the proportion in each column. See that 62% of the low wage jobs correspond to industrial, and 38% correspond to information.

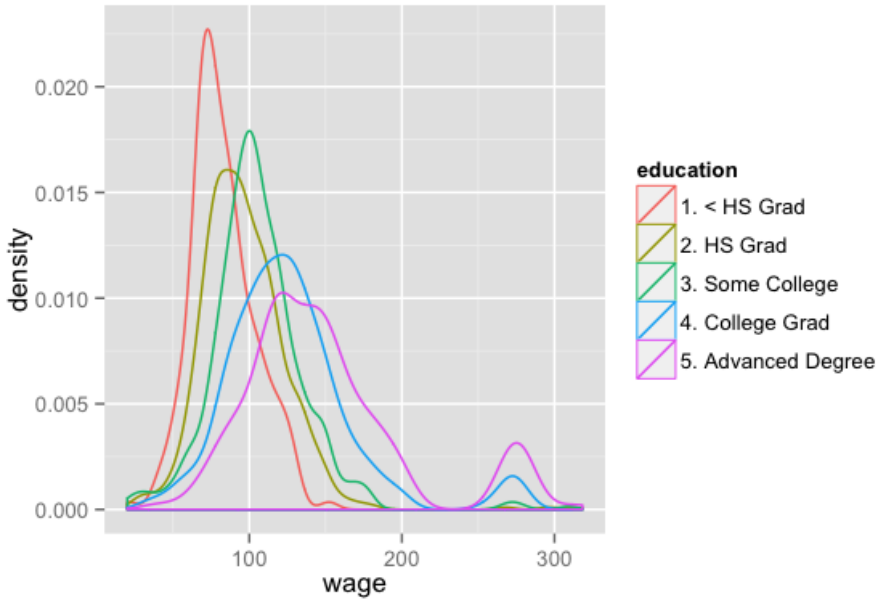
```
prop.table(t1,1)  # proportions 1 = by row, 2 = by column
```

```
cutWage         1. Industrial 2. Information
  [ 20.1, 91.7)        0.6207         0.3793
  [ 91.7,118.9)        0.5034         0.4966
  [118.9,318.3]        0.3908         0.6092
```

Density Plots

Use density plots to plot the values of continuous predictors using the **qplot** function here plotting the wage variable versus education in a density plot. This shows where the bulk of the data is. The x axis is wage and the y axis the proportion of the variable that falls into that bin of the x axis. The high school grads tend to have more values that are down in the lower part of the wage range and the advanced degree folks tend to be a little bit higher. There's also a group on the right that tends to be very high for both the advanced degree as well as the college grads which is shown in blue. Density plots can show things that box plots don't. It's also easier to overlay multiple distributions with density plots. In other words, if you break things up into a bunch of different groups, and you want to see how all the distributions change by group, density plots are useful.

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



Notes and further reading

Make your plots only in the training data. The test set should not be used for exploration. That is the same as training your model on the test set which will lead to overfitting. Look for in these plots is imbalance in the outcomes of the predictors. If all of the predictors tend to be one value in the one outcome group, and not another outcome group, then that's a good predictor.

But if there are only have three of one outcome and 150 of the other outcome that means it's going to be very hard to build an accurate classifier between those two classes.

Look for outliers or weird groups outlying the data that might suggest that there are some variables missing and groups of points that are not explained by any of the predictors.

Skewed variables which might be transformed and make look more normally distributed for things like regression models. But that may not matter as much as if you're using more of machine learning methods.

For more information on plotting in general you can look at the ggplot2 tutorial. You could also take the exploratory data analysis class in this data science specialization or you can look at the caret visualization tutorial. That will give you more information about useful prediction specific plots.

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

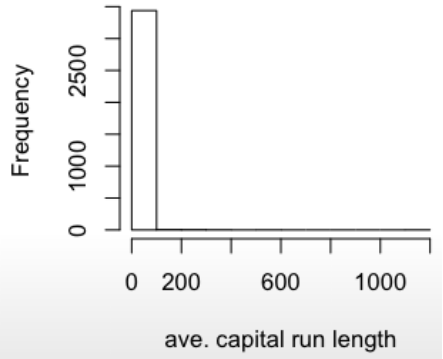
PML_2-5 Basic Preprocessing

This lecture is about preprocessing predictor variables. Plot the variables upfront to see if there's any weird behavior of those variables. They might need to be transformed in order to make them more useful for prediction algorithms. This is particularly true when you're using model based algorithms like linear discriminate analysis, naïve Bayes, and linear regression. Keep in mind that pre-processing can be more useful often when you're using model based approaches than when you're using more non-parametric approaches.

Why Preprocess?

Load the caret package, the kernlab package, and then attach the spam data. Only look at the training set. Split data right away into training and testing data. Now look at the spam data as we're trying to predict whether the data is spam or if it's good emails, ham. The variables are things like: 1) how many capitals do we see in a row, 2) hat's the run length for the number of capitals in a row in an email? Make a histogram of those values and see, for example, that almost all of the capital run links are very small but there are a few that are much, much larger. This is an example of a variable that is very skewed so it's hard to deal with in model based predictors. As a result you might want to preprocess the data.

```
library(caret); library(kernlab); data(spam)
inTrain <- createDataPartition(y=spam$type,
                               p=0.75, list=FALSE)
training <- spam[inTrain,]
testing <- spam[-inTrain,]
hist(training$capitalAve,main="",xlab="ave. capital run length")  # average capitals
```



The mean of this variable is about 4.7 but the standard deviation is much larger - much more highly variable.

```
mean(training$capitalAve)
```

```
[1] 4.709
```

```
sd(training$capitalAve)    # very large standard deviation highly skewed
```

```
[1] 25.48
```

Standardizing (Normalization)

Preprocessing will help avoid the machine learning algorithms getting tricked by the data that is skewed and highly variable. One method is basically standardizing variables. The usual way to standardize variables is to take the variable values and subtract their mean. Then subtract the mean and divide that whole quantity by the standard deviation. When you do that the mean of the standardized variables will be zero and the standard deviation will be one reducing a lot of the variability. And it will, standardize the variable somewhat.

```
trainCapAve <- training$capitalAve
trainCapAveS <- (trainCapAve  - mean(trainCapAve))/sd(trainCapAve) # convert to z-scores
```

```
mean(trainCapAveS)
[1] 5.862e-18

sd(trainCapAveS)
[1] 1
```

Standardizing - test set

Be aware that we can only use parameters that we estimated in the training set when we apply a prediction algorithm to the test set. ***When applying this same standardization to the test set, use the mean from the training set, and the standard deviation from the training set, to standardize the testing set values.*** What does this mean? It means that when doing the standardization, the mean will not be exactly zero in the test set. And the standard deviation will not be exactly one, because they're standardized by parameters estimated in the training set. Ideally they'll be close to those values even though we're using not the exact values built in the test set.

```
testCapAve <- testing$capitalAve
testCapAveS <- (testCapAve  - mean(trainCapAve))/sd(trainCapAve)
# note train, test sets and (testCapAve - mean(trainCapAve)) is divided by sd(trainCapAve)
```

```
mean(testCapAveS)
[1] 0.07579

sd(testCapAveS)
[1] 1.79
```

Standardizing - *preProcess* function

Use the **preProcess** function built into the caret package to do a lot of standardization. Pass it all of the training variables *except the actual outcome that we care about, the 58th in the data set*. With **method=c("center","scale"))** center and scale every variable. That does the same transformation discussed above where you subtract the mean and divide the result by the standard deviation. Note the mean of the variable **capitalAve** , it's the same as before - the mean is zero and the standard deviation is one. Preprocess can be used to perform a lot of the preprocessing techniques without being explicitly done as above.

```
preObj <- preProcess(training[,-58],method=c("center","scale")) # outcome=var 58
# method does same at normalization and creates the standardized predictors data set
trainCapAveS <- predict(preObj,training[,-58])$capitalAve # preProcess predictors, capitalAve
```

```
mean(trainCapAveS)
[1] 5.862e-18

sd(trainCapAveS)
[1] 1
```

Standardizing - *preProcess* function (on test set)

Use the object that's created using the preprocessing technique to apply that same preprocessing to the test set. Here the **preObj** is the object created with the preprocessing the training set. Pass the testing set to the predict function then will take the values calculated on the preprocessing object and apply them to the test set object. Again in the pre, post process test

set data the mean won't exactly be equal to zero for any variable and the standard deviation won't exactly be equal to one, but they'll be close, because the training set values were used to normalize.

```
testCapAveS <- predict(preObj,testing[,-58])$capitalAve  # note preObj, test set
```

```
mean(testCapAveS)
[1] 0.07579

sd(testCapAveS)
[1] 1.79
```

Standardizing - *preProcess* argument

You can pass the preprocessed commands directly to the train function in caret as an argument. For example we can send to the preprocessed argument of the train function, the command **preProcess=c("center","scale"**), that will center and scale all of the predictors before using those predictors in the prediction model. Centering and scaling is one approach, that will take care of some the problems that we see in these data. You can remove, very strongly biased predictors or predictors that have super high variability.

```
set.seed(32343)
modelFit <- train(type ~.,data=training,
                  preProcess=c("center","scale"),method="glm")
# center/scale predictors before using in model
modelFit
```

```
3451 samples
  57 predictors
   2 classes: 'nonspam', 'spam'

Pre-processing: centered, scaled
Resampling: Bootstrap (25 reps)

Summary of sample sizes: 3451, 3451, 3451, 3451, 3451, 3451, ...

Resampling results

  Accuracy  Kappa  Accuracy SD  Kappa SD
  0.9       0.8    0.007        0.01
```

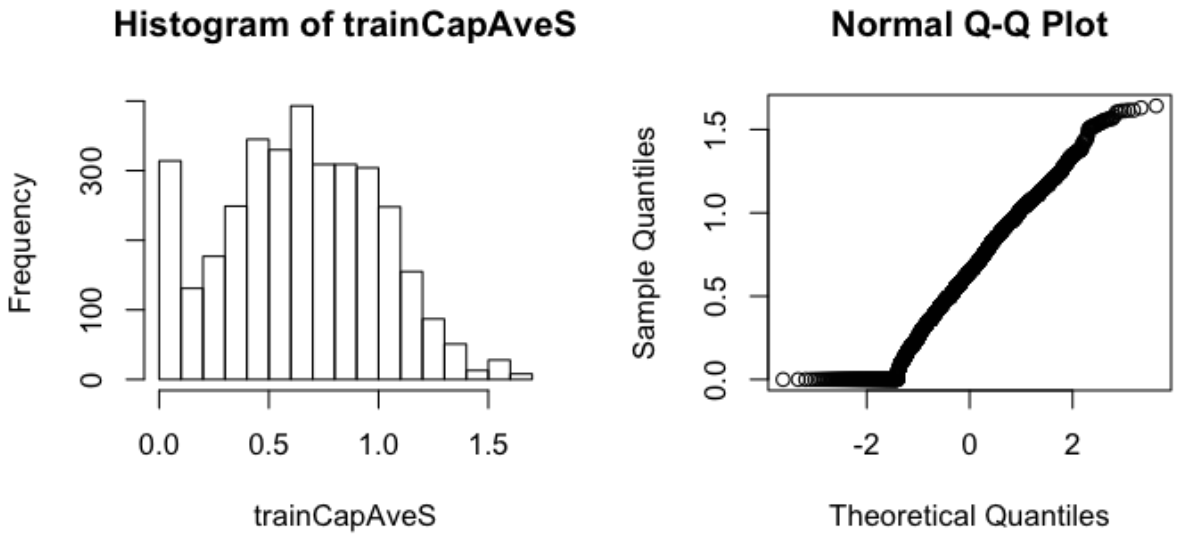
Standardizing - Box-Cox transforms (maximum likelihood)

You can use other transformations and one example is the box cox transforms. Box cox transforms are a set of transformations that take continuous data, and try to make them look like normal data. They estimate a specific set of parameters using maximum likelihood. So use the preprocess function and tell it to perform box cox transformations on each of the variables, then use **predict** with each of the different variables using that preprocess object, **preObj**, on the training set. I can look at the capital average values, and I can make a histogram of those values.

```
preObj <- preProcess(training[,-58],method=c("BoxCox")) # make cont. data look normal  - Box Cox
doesn't take care of a lot of repeated values
trainCapAveS <- predict(preObj,training[,-58])$capitalAve
par(mfrow=c(1,2)); hist(trainCapAveS); qqnorm(trainCapAveS)
```

Recall the original histogram that had a huge pile at zero and a few large values. Now it is a little bit more like a normal distribution, a bit more like a bell curve, but it doesn't take care of all of the problems. So, for example there's still a set of

values here at zero and in the Q-Q plot. This shows the theoretical quantiles of the normal distribution versus the sample quintiles that were calculated for our preProcess data. They don't perfectly line up. There is a group at the bottom that doesn't lie on a perfect 45 degree line. When a bunch of values are exactly equal to zero, this continuous transform doesn't take care of values that are repeated. So, it doesn't take care of a lot of the problems that would occur with using a variable that's highly skewed.




```
# compare histogram to earlier one with heavy skew.  Note QQ – from repeated values.
```

Standardizing - Imputing data

You can also impute data for these data sets as it's very common to have missing data. When there is missing data in the data sets, the prediction algorithms often fail. Prediction algorithms are not built to handle missing data in most cases. Impute missing data using k-nearest neighbor imputation. Set the seed again, because this is a randomized algorithm and we want reproducible results. Take just the capital average values and create a new variable called **CapAve**. Randomly generate a bunch of values using the **rbinom** function to set them equal to NA to set those values to be missing. The variable **capAve** is the **capitalAve** variable with a subset of values that are missing. How would I handle those missing values?

```
set.seed(13343)
```

Take just the capital average values and create a new variable called **CapAve**. Randomly generate a bunch of values using the **rbinom** function to set them equal to NA to set those values to be missing. The variable **capAve** is the **capitalAve** variable with a subset of values that are missing.

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

What follows is how to handle missing values in a dataset. Use the **preProcess** function and tell it to do k-nearest neighbor imputation, **method="knnImpute"**. If k equals ten, then the 10 nearest data vectors that look most like the data vector with the missing value and average those values to replace the variable that is missing. Then we can predict on our training set, all of the new values, including the ones that have been imputed with the k-nearest imputation algorithm.

```
# Impute and standardize by k nearest neighbors
preObj <- preProcess(training[,-58],method="knnImpute")
capAve <- predict(preObj,training[,-58])$capAve
```

Standardize those values, using the same standardization procedure as before, by subtracting the mean and dividing the result by the standard deviation.

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

Compare between the actual when we set some of the values to be equal to NA in advance and were then imputed withthe values that were truly there before we removed them and made them NAs. Most of the differences are very close to zero

```
# compare actual values replaced with imputed values
quantile(capAve - capAveTruth)
```

```
        0%         25%         50%         75%        100%
-1.1324388 -0.0030842 -0.0015074 -0.0007467  0.2155789   # most close to zero
```

Look just the values that were imputed – the capAve quantile of the difference between the imputed values and the true values but only for the ones set to NA. Most values are close to zero, but clearly some are more variable than before.

```
# only look at ones that were missing via selectNA index
quantile((capAve - capAveTruth)[selectNA])
```

```
        0%         25%         50%         75%        100%
-0.9243043 -0.0125489 -0.0001968  0.0194524  0.2155789
```

And then you can look at the ones that were not the ones that we selected to be NA and see that they're even closer to each other. The ones that got imputed are a little bit further apart - but not that much further apart.

```
quantile((capAve - capAveTruth)[!selectNA])
```

```
        0%         25%         50%         75%        100%
-1.1324388 -0.0030033 -0.0015115 -0.0007938 -0.0001968
```

Notes and further reading

There is much more to learn about training and testing sets in terms of transformations.

When dealing with factor variables it's a little bit more difficult to know what the right transformation is. Most machine learning algorithms are built to deal with either binary predictors, in which the binary predictors are not pre-processed. Or continuous predictors in which case it's often expected that the data are preprocessed to look more normal. You can go to this link that I've linked here to look, into more information about how to preprocess data for prediction using the caret package.

- Training and test must be processed in the same way
  Training and test sets must be processed in the same way. The caret package handles a lot of this in the sense that when you train a data set using ***preProcess*** *functions, built into the* ***train*** *function in caret, the way it applies that preprocessed function to the test set it will handle the preprocessing in the correct way. In general, anything you do to the training set will create a set of parameters. Use only those parameters when you apply it to the test set. Don't estimate new transformations on the test set alone.*

- Test transformations will likely be imperfect.
    - Especially if the test/training sets collected at different times
      And that means the test set transformations will likely be imperfect, especially if the test and training sets are collected at different times or in different ways. Some of the transformations won't necessarily work as well. The transformations I'm talking about, so far other than imputation are based on continuous variables.
- Careful when transforming factor variables!
  When dealing with factor variables it's a little bit more difficult to know what the right transformation is. Most machine learning algorithms are built to deal with either binary predictors in which the binary predictors are not pre-processed or continuous predictors in which case it's often expected that the data are preprocessed to look more normal.
- preprocessing with caret
  Use this URL for more information about how to preprocess data for prediction using the caret package.

PML_2-6 Covariate Creation

`Covariates : predictors or features`

Covariates are sometimes called predictors and sometimes called features. They're the variables that you include in your model to combine them to predict the outcome that you care about. There are two levels of covariate creation, or feature creation. The first level is, taking the raw data and turning it into a predictor that you can use. The raw data often takes the form of an image, or a text file, or a website. This information is very hard to build a predictive model around if it hasn't been summarized in some useful way into either a quantitative or qualitative variable. We want to do is take that raw data and turn it into features or covariates which are variables that describe the data as much as possible while giving it some compression and making it easier to fit standard machine-learning algorithms.

Level 1: From raw data to covariate (features / predictors)

HI

WE'VE DISCOVERED YOU ARE THE HEIR TO AN INCREDIBLE FORTUNE. PLEASE SUBMIT YOUR NAME, ADDRESS AND BANK ACCOUNT SO WE CAN SEND YOU $$$$$$$$.

JOE JOHNSON

capitalAve	you	numDollar	...
1	2	8	...

So the idea here is, so suppose you have a email, this is an email, an example email here on the left. And so it's very hard to plug the email itself into a prediction function, because most prediction functions are based on the idea of taking a small number of variables and building a quantitative model around them and it doesn't work for a free text for example. So the first thing that you need to do is create some features, and those features are just variables that describe the raw data. So in this case, in the case of an email, we might think of different ways that we could describe this email. For example, when I calculate the average number of capitals that are in the email, in this case 100% of the letters in the email are capital letters, you might say what's the frequency a particular word appears. So for example, you might say, how often does you appear? And you appears twice in this email, so we say that we calculate two for this email. That's a feature. You might also calculate the number of dollar signs. This might be a really good predictor of whether an email is spam or not. And so here you see there are a large number of dollar signs, there are eight of them, so we calculated another feature of that data set. So this step, the raw data of the covariate, usually involves a lot of thinking about the structure of the data that you have and what is the right way to extract, extract the most useful information in the fewest number of variables that captures everything that you want. The next stage is transforming tidy covariates. In other words, we calculated this number, say capital average, the average number of capitals in the data set. But it might not be the average number that's related very well to the outcome that we care about, it might be the average number of capitals squared or cubed, or it might be some other function of that.

And so the next stage is transforming the variables into sort of more useful variables. So for example, if we load the kernlab data and the spam data set, we can take the capital average, so the, this is basically this variable right here, the fraction of letters that are capitals. And we could square that number, and assign it to a new variable, capital average squared, that might be useful later in our prediction algorithm.

Level 2: Transforming tidy covariates

```
library(kernlab);data(spam)
spam$capitalAveSq <- spam$capitalAve^2   # example of using a transform on capitalAve
```

So those are the two steps in creating covariates. So the first step the raw data, the covariate really depends heavily on the application. So like I showed you on the previous slide, in an email case, it

might be extracting the fraction of times a word appears or something like that. In a case of voice, it might be knowing something about the frequency or the timbre of which voices are typically fall. In the case of images, it might be identifying features of the images. So if it's faces, where are the noses or the ears or the eyes are? And it will depend greatly what your application is. And the balancing act here is definitely summarization versus information loss. In other words, it, the, the best features are features that capture only the relevant information in, say, the image or the email, and throw out all the information that's not really useful at all. And so the idea is that you have think very carefully about how to pick the right features that explain most of what's happening in your raw data. So some examples here, for text files, it might be the frequency of words or frequency of phrases. There's this cool site, Google ngrams, which tells you about the frequency of different phrases that appear in books going back in time. For images, it might be edges and corners, blobs and ridges for example. These are all ideas about how do you identify different structures in an image. For websites it might be the number and type of images, where buttons are, colors and videos. This is a huge area of importance in web development which is called A/B testing, which is called randomized trials and statistics, which is basically showing different versions of a website with different values of these different features and predicting which one will introduce a more clicks or get more people to buy products. For people you can imagine features of people are their height, weight, hair color, etc. It's basically any summary of the raw data that you can make as a potential feature. And often this involves quite a bit of scientific thinking and business acumen to know what the right covariates are for a particular problem. So the more knowledge you have of a system, the better job you'll do at feature extraction in general. In general it's a good idea to have a really clear understanding of why this set of data is useful for, to predicting the outcome you care about. So there's this balance between summarization and information loss, and in general, it's better to err on the side of creating more features. You lose less information and then filter some of those features out during your model-building process. This can all be automated and has been automated in various different ways, but you generally have to use a lot of caution when using that approach because sometimes a particular feature will be very useful in the training set that you created but won't be very useful in a new set of data and the test set won't generalize well. So the second level is taking tidy covariates, so these are features you've already created on the data set, and then creating new covariates out of them. Usually this is transformations or functions of the covariates, that might be useful when building a prediction model. This can sometimes be more necessary for methods like, regression, or support vector machines that might depend a little bit more on what the distribution of the data are, and a little bit less for things like classification trees, where the idea here is you don't necessarily have as much odel-based prediction. In other words, you don't depend quite so much on the data looking a particular way. On the other hand, in general, it's a good idea to spend some time making sure you have the right covariates in your model. So you, when you create these functions or decide on these functions, you have to do it only in the training set. This is a common theme of machine learning. Building features can only happen in the training set, it can't happen in the test set. Later when you apply your prediction to your function to the test set, you will make that same function of the covariate so you can apply your predictor. But the original creation or thinking about what covariates to build has to happen only in the training set, otherwise you'll lead to overfitting. And the best approach I've found is through exploratory analysis, so basically making plots and making tables of the data, and trying to understand what are the patterns of variation in your data set and how they might relate to the outcome. When you're using the care package or doing this analysis in r, the new covariates need to be added to data frames so that they can be used in downstream prediction. And it's important to make sure that the names of the new variables are recognizable so that you can use the same name on your testing data set.

Level 1, Raw data -> covariates

- Depends heavily on application
- The balancing act is ***summarization vs. information loss***
- Examples:
    - Text files: frequency of words, frequency of phrases (Google ngrams), frequency of capital letters.
    - Images: Edges, corners, blobs, ridges (computer vision feature detection))
    - Webpages: Number and type of images, position of elements, colors, videos (A/B Testing)

  - People: Height, weight, hair color, sex, country of origin.
- The more knowledge of the system you have the better the job you will do.
- When in doubt, err on the side of more features
- Can be automated, but use caution! *[may work well on training set but not on test set]*

Level 2, Tidy covariates -> new covariates

- More necessary for some methods (regression, svms) than for others (classification trees). SVMs – support vector machines or support vector networks
- Should be done only on the training set
- The best approach is through exploratory analysis (plotting/tables)
- New covariates should be added to data frames

Load example data

Load in this data from the ISLR package, the caret package and attach the data on wages. Build a training and test sets. Do the covariate creation in the training set. Use the create data partition function to create the index, indices for the two data sets. Then separate the data into a training set and into a test set.

```
library(ISLR); library(caret); data(Wage);
inTrain <- createDataPartition(y=Wage$wage,
                               p=0.7, list=FALSE)  # creates indices for train set
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
```

Common covariates to add, dummy variables

Basic idea - convert factor variables to [indicator variables]

In statistics and econometrics, particularly in regression analysis, a dummy variable (also known as an indicator variable, design variable, Boolean indicator, categorical variable, binary variable, or qualitative variable) is one that takes the value 0 or 1 to indicate the absence or presence of some categorical effect that may be expected to shift the outcome

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

```
 1. Industrial 2. Information
          1090           1012
```

One very common idea when building machine learning algorithms is to turn covariates that are qualitative, or factor variables, into what are called dummy variables. The basic idea is suppose we have a variable, in this case let's look in the training set at the variable called job class. So that job class has two different levels,. We could try to plug that variable directly into a prediction model, but the values of that variable are actually a set of characters – 'industrial" or "information". Is is sometimes hard for prediction algorithms to use those qualitative information variables in order to actually do the prediction. So turn it into a quantitative variable and with the caret package it is the dummy variables function, **`dummyVars()`**. It passes in a model so the outcome is wage with job class the predictor variable and the training set is where we're going to be building those dummy variables.

```
dummies <- dummyVars(wage ~ jobclass,data=training) # wage = output; jobclass=predictor
```

Apply the predict function, the dummies object and a new data set **`newdata= training`** data set, you get, two new variables out. So the first is an indicator that you are industrial, and the second is an indicator that you're information. A one means that column is your job classification, zero means not your job classification. There are only two different levels of this variable - industrial and information. But if you had three variables here, every column would have two zeros, because those

are the two classes you don't belong to, and a one for the class that you belong to. So this is taking these factor or qualitative variables and turning them into quantitative variables.

```
head(predict(dummies,newdata=training))
# predict creates two new variables.  Turns factor variables into quantitative – binary variables.
```

```
       jobclass.1. Industrial jobclass.2. Information
231655                       1                       0
86582                        0                       1
11141                        0                       1
229379                       1                       0
86064                        1                       0
```

Removing zero covariates

Some variables basically have no variability in them. If you create a feature for emails thattests for any letters in it at all, every single email will have at least one letter in it, so that variable will always be equal to true. Thus it has no variability and it's probably not going to be a useful covariate. Use is this near-zero variable function in caret to identity those variables that have very little variability and will likely not be good predictors. Apply it to the training data set data frame. You can save the metrics to show what the variables are and the variance level. The example displays the percentage of unique values for a particular variable, and in this case the variable year has about 0.33% unique values, and it's not a near zero variable or a near zero variance variable. The variable sex is basically males so it has a very low frequency ratio. In other words, it's all one category, and ends up being a near zero variable. Use the nzv column of the matrix to throw out the variables sex and region that shouldn't be used in prediction algorithms. This is a way to throw remove less meaningful predictors right away.

```
nsv <- nearZeroVar(training,saveMetrics=TRUE)  # identify variables with little variance
nsv
```

```
           freqRatio percentUnique zeroVar   nzv
year           1.029       0.33302   FALSE FALSE
age            1.122       2.80685   FALSE FALSE
sex            0.000       0.04757    TRUE  TRUE
maritl         3.159       0.23787   FALSE FALSE
race           8.529       0.19029   FALSE FALSE
education      1.492       0.23787   FALSE FALSE
region         0.000       0.04757    TRUE  TRUE
jobclass       1.077       0.09515   FALSE FALSE
health         2.452       0.09515   FALSE FALSE
health_ins     2.269       0.09515   FALSE FALSE
logwage        1.198      17.26927   FALSE FALSE
wage           1.185      18.07802   FALSE FALSE
```

Sex and region are near zero variables and should be discarded as predictors.

Spline basis

Instead of fitting straight lines through the data with linear regression or generalized linear regression as your prediction algorithm, you can sometimes fit curvy lines. One way is with basis functions found in the splines package. The bs function will create a polynomial variable. Pass a single variable from the training set, age variable with a third degree polynomial for this variable, **df=3**. A three-column matrix is output with three new variables. The first column variable corresponds to age, the actual age values. There are scales for computational purposes. The second column will correspond to age squared. So, This fits a quadratic relationship between age and the outcome. The third column corresponds to age cubed to allow a cubic relationship between age and the outcome. Include these covariates in the model instead of just the age variable when fitting a linear regression to allow for curvy model fitting.

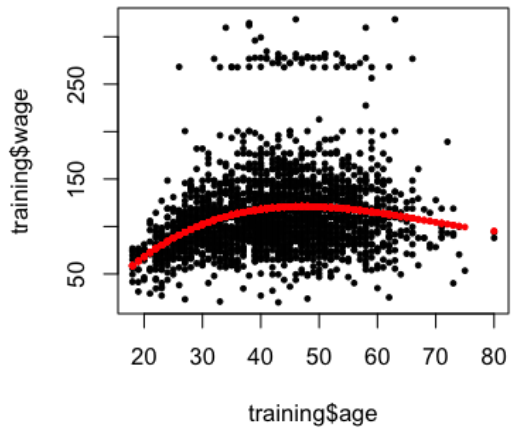
```
library(splines)
bsBasis <- bs(training$age,df=3) # polynomial variable – 3rd degree
bsBasis
```

```
               1          2           3     # 1 – age, 2 – age^2, 3- age^3
  [1,] 0.00000 0.0000000 0.000e+00
  [2,] 0.23685 0.0253768 9.063e-04
  [3,] 0.44309 0.2436978 4.468e-02
  [4,] 0.43081 0.2910904 6.556e-02
  [5,] 0.42617 0.1482327 1.719e-02
  [6,] 0.41709 0.1331149 1.416e-02
  [7,] 0.31823 0.0540390 3.059e-03
  [8,] 0.36253 0.3866940 1.375e-01
  [9,] 0.44436 0.2275981 3.886e-02
 [10,] 0.20449 0.0179375 5.245e-04
 [11,] 0.07768 0.3601465 5.566e-01
 [12,] 0.13145 0.0066841 1.133e-04
 [13,] 0.39290 0.1042387 9.218e-03
 [14,] 0.26654 0.0339238 1.439e-03
```

Fitting curves with splines

Here fit a linear model with wage as the outcome. Again the tilde tells you what's we're predicting it with **bsBasis**. That is we pass it all the predictors that we generated from the polynomial model - age, age squared, and age cubed. Then plot the age data versus the wage data; age on the x axis, wage on the y axis. The result is a curvilinear relationship between these two variables. And so we can plot age and the predicted values from our linear model, including the polynomial terms and you get a curve fit through the data set as opposed to a straight line. This is one way that to generate new variables by allowing more flexibility in the way that you model specific variables.

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



Splines on the test set

On the test set predict those same variables. The idea that's incredibly critical for machine learning is when you create new covariates. Create the covariates on the test data set using the exact same procedure used on the training set. Do that by predicting, **predict()**, from the variable created using the **bs()** function, **bsBasis**, a new set of values with the testing set age values, **age=testing$age**. These are the values to actually plug in to the prediction model when testing it out on the test set – as opposed to creating a new set of predictors based on applying the **bs()** function applied directly to the age variable. This would create a new set of variables on the test set that isn't related to the variables created on the training set and may introduce some bias.

```
predict(bsBasis,age=testing$age)  # have to predict same values on test set!
```

```
            1         2         3  # these are same as bsBasis!
  [1,] 0.00000 0.0000000 0.000e+00
  [2,] 0.23685 0.0253768 9.063e-04
  [3,] 0.44309 0.2436978 4.468e-02
  [4,] 0.43081 0.2910904 6.556e-02
  [5,] 0.42617 0.1482327 1.719e-02
  [6,] 0.41709 0.1331149 1.416e-02
  [7,] 0.31823 0.0540390 3.059e-03
  [8,] 0.36253 0.3866940 1.375e-01
  [9,] 0.44436 0.2275981 3.886e-02
 [10,] 0.20449 0.0179375 5.245e-04
 [11,] 0.07768 0.3601465 5.566e-01
 [12,] 0.13145 0.0066841 1.133e-04
 [13,] 0.39290 0.1042387 9.218e-03
 [14,] 0.26654 0.0339238 1.439e-03
 [15,] 0.20449 0.0179375 5.245e-04
 [16,] 0.29109 0.4308138 2.125e-01
```

Notes and further reading

Here are some ideas and future reading for you, Level one feature creation is basically all about science or, application specific knowledge, I've found that the best way to do it is Googling feature extraction for the type of data that you're trying to analyze. Feature extraction for images. Feature extraction for voice. You can also just look up that particular data type and see as much information as you can about it. In particular you're looking for what are the salient characteristics that are likely to be different between individual samples. In general you want to err on the side of over-creation of features because you can always filter them out later in the machine learning algorithm process.

In some applications like images and voices, it's often both possible and pretty much necessary to create features that aren't necessarily just things that you imagine. It's very hard to know exactly what the right components of an image to include as features in a model, and so there are things like you may have heard of deep learning which is basically a way of creating features for things like images and voice. And this is a nice tutorial I've linked to that kind of explains how that feature creation process works for those things. But in general, automatic feature creation requires an equal level of thinking to make sure that the features being generated by your feature creation process make sense.

Level 2 feature creation covariates to new covariates can be done a lot with the **preProcess** components of the caret package. You can create new covariates using any of the functions in R if they make sense to you. The key is making lots of plots and doing exploratory analysis to see where the connections between the predictors and the outcome are. You can create new covariates if you think they will improve fit. Again, you can err on the side of over-creation of features, but sometimes features just are nonsensical and you shouldn't create them. Be careful about overfitting in the sense that if you create lots of features that are particularly good for just your training set, they may not work well in the test set. A good idea is when you over-create lots of features to do some filtering before you actually apply your machine learning algorithm.

The tutorial on preprocessing with caret is a good place to start for really basic preprocessing. And if you want a flit spline model, you can use the **gam** method in the **caret** package which allows smoothing of multiple variables using a different smooth for every variable.

And more on feature creation and data tidying is in the Getting and Cleaning Data course from the Data Science specialization.

- Level 1 feature creation (raw data to covariates)
    - Science is key. Google "feature extraction for [data type]"
    - Err on overcreation of features
    - In some applications (images, voices) automated feature creation is possible/necessary
    - http://www.cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf
- Level 2 feature creation (covariates to new covariates)
    - The function preProcess in caret will handle some preprocessing.
    - Create new covariates if you think they will improve fit
    - Use exploratory analysis on the training set for creating them
    - Be careful about overfitting!
- preprocessing with caret
- If you want to fit spline models, use the **gam** method in the **caret** package which allows smoothing of multiple variables.
- More on feature creation/data tidying in the Obtaining Data course from the Data Science course

PML_2-7 Preprocessing with Principal Components Analysis (PCA)

[basically trying to find (nearly) orthogonal predictor variables]

This is a lecture a lecture about preprossings, preprocessing covariants with principal components analysis. The idea is that often you have multiple quantitative variables and sometimes they'll be highly correlated with each other. In other words, they'll be very similar to being the almost the exact same variable. In this case, it's not necessarily useful to include every variable in the model. You might want to include some summary that captures most of the information in those quantitative variables.

Correlated predictors

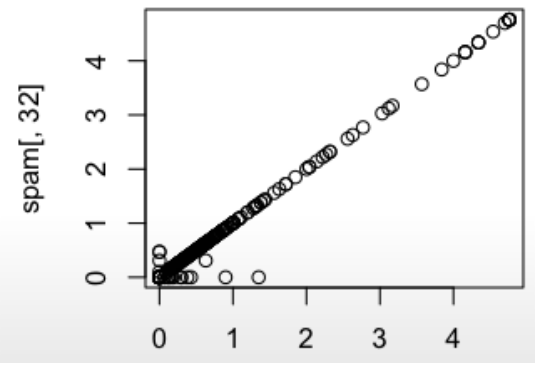
Load **caret**, the **kernlab** package, and attach the spam data set. Create training and test sets and perform only operations on the training set. All exploration and model creation and feature building has to happen in the training set. Then Leave out the 58th column of this training set which is the outcome and include all the other predictor variables. Calculate the correlation between all predictor variables and take the absolute value. I'm looking for all the predictor variables that that have a very high correlation or are very similar to each other. Each variable has a correlation of 1 with itself. I'm not interested in those variables and remove them by setting the diagonal of the correlation matrix to 0. That's basically just setting the correlation between variables with itself, equal to 0. Then which of the variables have a high correlation with each other? Which of the variables have a correlation greater than 0.8? Two variables have a very high correlation with each other. They are the num415 and num857. If the numbers 415 and 857 appears in the email and frequently appears together, it is likely because there's a phone number that has similar variables there. So, if I look at the spam dataset, at the columns 34 and 32, which I got from getting that from the previous correlation variable. I see that it's these two variables, these two columns that are highly correlated with each other. And if I plot those two columns against each other, I see exactly what I'd expect. So, the frequency of 415 and 857 is incredibly highly correlated, this basically lie perfectly on a line with each other. As the number of 415 appears more frequently, so does the number 857. So the idea is that including both of these predictors in the model might not necessarily be very useful.

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

M <- abs(cor(training[,-58]))  # abs[covariate correlation (less outcome, variable 58)]
diag(M) <- 0   # self-correlation (diagonal)= 1, set to zero for next step.
which(M > 0.8,arr.ind=T)  # find highly correlated variables
```

```
       row col
num415  34  32
num857  32  34
```

Correlated predictors

```
names(spam)[c(34,32)]
[1] "num415" "num857"

plot(spam[,34],spam[,32])  # confirm high correlation
between num415 and num857
```



Basic PCA idea

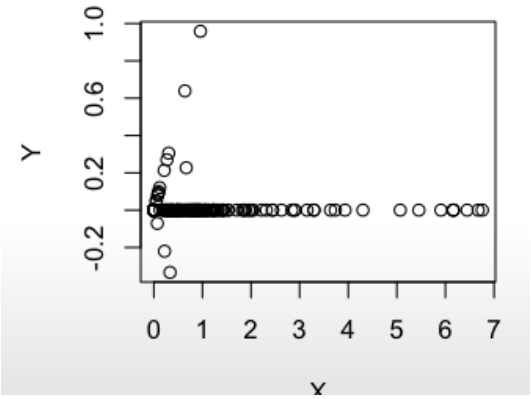
- We might not need every predictor
- A weighted combination of predictors might be better
- We should pick this combination to capture the "most information" possible
- Benefits
    - Reduced number of predictors
    - Reduced noise (due to averaging)

The idea is how to take those variables and turn them into, say, a single variable that might be better? One idea is to use a weighted combination of those predictors that explains most of what's going on. The idea is to pick the combination that captures the most information possible. The benefits are that you're reducing the number of predictors you need to include in your model and you're also reducing noise. In other words, you're averaging or combining variables together, so you might reduce them together. So you do this in a clever way by doing principal component analysis. You're trying to figure out a combination of these variables that explain most of the variability. For example, here's a combination that takes 0.71 * num415 + 0.71 *num857 to create a new variable called x which is basically the sum of the two variables. Then I could take the difference of those two variables with 0.71 * num415 - 0.71 * num857. So x is adding the two variables together, y is subtracting the two variables. Then plot y versus x and so you can see most of the variability is happening in the x-axis. There's lots of points all spread out across the x-axis with most of the points are clustered right at 0 on the y-axis. Almost all of these points have a y value of 0. Adding the two variables together captures most of the information in those two variables and subtracting the variables captures less information. This suggests using the sum of the two variables as a predictor. That will reduce the number of predictors that we will have to use and reduce some of the noise. So there are two related problems to how you do this in a more general sense. And so the ideas are find a new set of variables based on the variables that you have that are uncorrelated and explain as much variability as possible. In other words from the previous plot, we're looking for the x variable which has lots of variation in it. And not the y variable which is almost always 0. So if you put the variables together in one matrix, create the best matrix with fewer variables. In other words this is lower rank if you're talking mathematically that explains the original data. These two problems are very closely related to each other. And they're both the idea that, can we use fewer variables to explain almost everything that's going on. The first goal is a statistical goal and the second goal is a data compression goal. But they're also, they're both very useful for machine learning.

We could rotate the plot [find out where the info is]

$$X = 0.71 \times num415 + 0.71 \times num857$$
$$Y = 0.71 \times num415 - 0.71 \times num857$$

```
X <- 0.71*training$num415 + 0.71*training$num857
Y <- 0.71*training$num415 - 0.71*training$num857
plot(X,Y)
# suggests using the weighted sum of num415 and num857 as the replacement predictor
```



Related problems

You have multivariate variables X1, …, Xn so X1 = (X11, …, X1m)

- 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 matrix) that explains the original data.

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

So there are two related problems to how you do this in a more general sense. And so the ideas are find a new set of variables based on the variables that you have that are uncorrelated and explain as much variability as possible. In other words from the previous plot, we're looking for the x variable which has lots of variation in it, not the y variable which is almost always 0. So if you put the variables together in one matrix, create the best matrix with fewer variables. In mathematical terms, this is a lower rank that explains the original data. These two problems are very closely related to each other in the sense we can use fewer variables to explain almost everything that's going on. The first goal is a statistical goal and the second goal is a data compression goal, both very useful for machine learning.

Related solutions - PCA/SVD

There are two related solutions, very similar to each other. X is a matrix with a variable in each column, and an observation in each row, like a data in frame you usually have in R. Then the singular value decomposition is a matrix decomposition. So it takes that data frame X and breaks it up into three matrices, a U matrix, and D matrix, and a V matrix. The columns of U are called the left singular vectors and the columns of V are called the right singular vectors. D is a diagonal matrix. Those are called the singular values. You will learn about this in getting data or exploratory data analysis if you've taken those classes. ***The principle components are equal to the right singular vectors if you scale the data in the same way.*** In other words, the solution to both of those problems that I talked about on the previous slide is the same if you do the right scaling. The idea is the variables in V are constructed to explain the maximum amount of variation in the data.

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"

$$X = U D V^T$$

where the ***columns of U*** are orthogonal (left singular vectors), the columns of ***V are orthogonal*** (right singluar vectors) and ***D is a diagonal*** matrix (singular values).

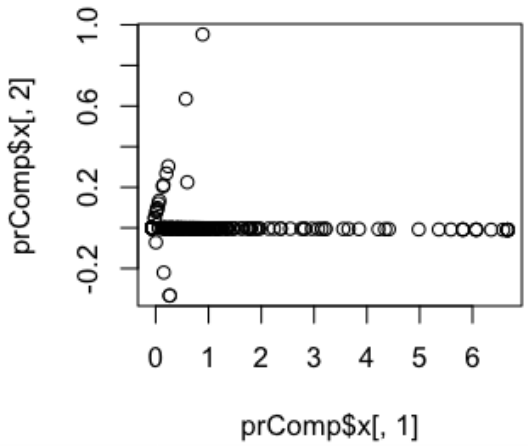
PCA – Principal Components Analysis

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*

Take the spam data set and just take those two variables that were highly correlated with each other, variables 34 and 32. Then do principal components same as the singular value decomposition on the small data set that just consists of the two variables. If we plot the first principle component versus the second principle component we see a plot that is very similar to the one that I showed you earlier where the first principal component looks like adding the two variables together and the second principal component looks a lot like subtracting the two variables from each other. So why would we do principal components instead of just adding and subtracting? Well, principal components allow you to perform this operation with more than just two variables. You may be able to reduce all of the variables down into a very small number of combinations of sums and differences and weighted sums and differences of the variables that you've observed. Principal components can evaluate a large number of quantitative variables and reduce the number quite a bit.

```
smallSpam <- spam[,c(34,32)]
prComp <- prcomp(smallSpam)  # finds principal components, allows working with > 2
plot(prComp$x[,1],prComp$x[,2])
```



You can also look at the rotation matrix with this principal component object which is basically how it's summing up the two variables to get each of the principal components. You can see why I put 0.71 in the sum and the difference on the first slide. So, principal component one is just Principal component two is just the difference 0.7061 * num415 - 0.7081 * num857. In this particular case the first principal component, the one that explains the most variability is just adding the two variables up and the variable that explains the second most variability in these two variables is the taking the difference between the two variables. So in this spam data we can do this for a more variables than just the two variables. This is why principal components may be useful.

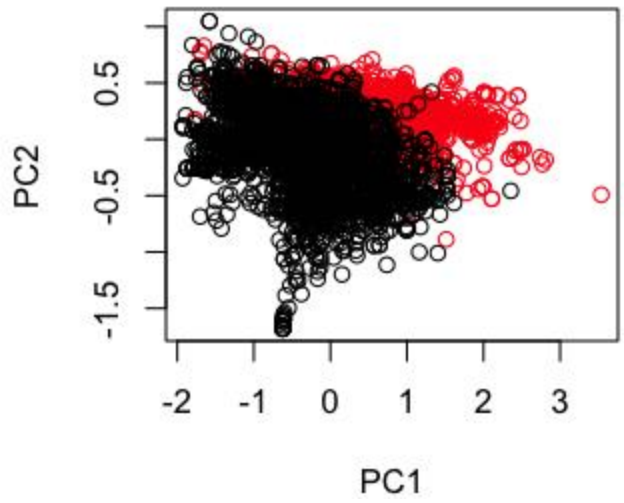
```
prComp$rotation  # shows weighting coefficients of variables, in order of variability
PC1 is more influential that PC2
```

```
           PC1     PC2
num415 0.7081  0.7061
num857 0.7061 -0.7081
```

PCA on SPAM data

Create a variable that's just going to be the color we're going to color our points by. Its color equals black if not a spam and color equal to red if spam. The statement **prcomp()** calculates the principal components on the entire data set. Note the $\log_{10}$ transform plus 1 applied to the data set to make the data look a little bit more Gaussian. Some of the variables are normal looking and some of the variables are skewed. You often have to do that for principal component analysis to look sensible. Then calculate the principal components of the entire data set. So in this case I can now again plot principal component one, versus principal component two. Principle component one is no longer a very easy addition of two variables. It might be some quite complicated combination of all the variables in the data set. But it's the combination that explains the most variation in the data. Principle component two is the combination that explains the second most variation. Principal component three explains the third most and so forth. Plot principal component one, the variable that's calculated versus principal component two, another variable that I's calculated, then color them by the spam indicator. Each of these points corresponds to a single observation. The red ones correspond to spam observations and the black ones ham observations. You could see that in principal component one space, along the principal component one axis, there's a little bit of separation of the ham messages from the spam messages. In other words the spam messages tend to have a little bit higher values than principal component one.

```
typeColor <- ((spam$type=="spam")*1 + 1)  # color identification of data classification
prComp <- prcomp(log10(spam[,-58]+1))  # log transform to 'normalize', note +1!
plot(prComp$x[,1],prComp$x[,2],col=typeColor,xlab="PC1",ylab="PC2")
```



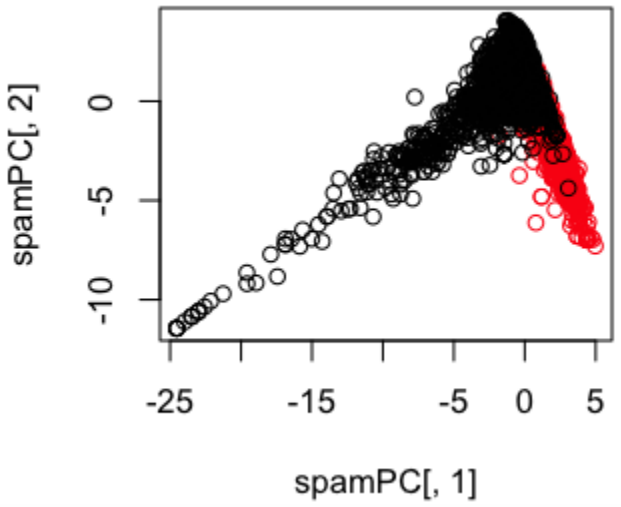
This is a way to reduce the number of data set variables while still capturing a large amount of variation with this idea behind feature creation.

```
# SPAM / HAM separation evident in PC1 axis &
not PC2
```

PCA with caret

Do this in **caret** in a similar type operation with a caret package, the pre-process function. Use the pre-process function with the same data set as before. Specify the method to use (n this case principal component analysis or **pca**) and the number of principal components to compute. Then calculate the values of each new principle component so the principle components are two variables. There is principle component one, principle component two. And they're basically a model that you fit to the data. So the idea is that if you get a new observation you have to predict what the principle component will look like for that new variable. So we pass this pre-processed object and the data set to the **predict()** function for the principle components. Plot them spam[, pc 1], principle component 1, versus principle component 2. Note a bit of separation between the ham and the spam messages in both principle component one and principle component two.

```
preProc <- preProcess(log10(spam[,-58]+1),method="pca",pcaComp=2)
# calculate values of each new principle component, two variables, a model that you fit
to the data.  If you get a new observation you have to predict what the principle component will
look like for that new variable. So we pass this pre-processed object and
the data set, to the predict function and that gives us the principle component.
spamPC <- predict(preProc,log10(spam[,-58]+1))
plot(spamPC[,1],spamPC[,2],col=typeColor)
```




```
# SPAM /HAM separation evident in both PC1 &
PC2
```

Preprocessing with PCA

You can do this with preprocessing with the method PCA, using the **preProcess** function. Then you can create training predictions with the predict function. Next fit a model that relates the training variable to the principal component. Here I haven't used the full training set as the data for fitting my model. I've just passed it the principal components for the model fitting.

```
preProc <- preProcess(log10(training[,-58]+1),method="pca",pcaComp=2)
trainPC <- predict(preProc,log10(training[,-58]+1))
modelFit <- train(training$type ~ .,method="glm",data=trainPC)
```

For the test data set use the same principal component that was calculated in the training data set for the test variables. Pass the **preProc** object that we calculated in the training set but now we pass at the new **testing** data. So the **predict** function is going to take the principle components we calculated from training and get the new values for the test data set on those same principle components.

```
testPC <- predict(preProc,log10(testing[,-58]+1))
```

Then **predict** using the **modelFit** on the original data using the test principal components **testPC**. And you can use the **confusionMatrix** argument in **caret** to get the accuracy. And so, here we calculated a relatively small number of principal components but still have a relatively high accuracy in prediction. Principal component analysis can reduce the number of variables while maintaining accuracy. So the other thing that you can do is you

```
confusionMatrix(testing$type,predict(modelFit,testPC))
```

```
Confusion Matrix and Statistics

          Reference
Prediction nonspam spam
   nonspam     646   51
   spam         64  389

               Accuracy : 0.9
                 95% CI : (0.881, 0.917)
    No Information Rate : 0.617
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.79
 Mcnemar's Test P-Value : 0.263

            Sensitivity : 0.910
```

Alternative (sets # of PCs)

You can actually decide not use the predict function separately, instead build it right into the training exercise. Take the **train** function from the **caret** package and pass the training set. Specify **preProcess** with principal component analysis. It will do that pre-processing as part of the training process. Nest to do the prediction on the new data set you just pass it a testing data set and will , it will actually calculate the principal components.

The more elaborate way of calculating the PC's first and then passing them to the model shows what's going on under the hood when the **preProcess** command is passed to the **train** function in the **caret** package.

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

```
# confusion matrix now calculates the PC's.
confusionMatrix(testing$type,predict(modelFit,testing))
```

```
Confusion Matrix and Statistics

          Reference
Prediction nonspam spam
   nonspam     660   37
   spam         54  399

               Accuracy : 0.921
                 95% CI : (0.904, 0.936)
    No Information Rate : 0.621
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.833
 Mcnemar's Test P-Value : 0.0935

            Sensitivity : 0.924
```

Final thoughts on PCs

PCs are most useful for linear type models including linear discriminant analysis and linear and generalized linear regression. It can make it a little bit harder to interpret the predictors in the case with only two variables as it was just the sum and the difference of those variables so it was very easy to predict what that meant. In general though, if you do principal components on a large number of quantitative variables, each principal component might be quite a complex weighted sum of the variables you've observed and could be very hard to interpret.

You have to watch out for outliers which can really wreak havoc on calculating principal components. Check for outliers by looking at an exploratory analysis first and identify any outliers. Consider transforms like $\log_{10}$ transform or a Box Cox transformations of the data. And again, plotting predictors to identify problems is the key place to figure out where this is working out.

For more information, you can see the exploratory data analysis class where we talk about principal component analysis and SVD in more detail. And this book the Elements of Statistical Learning has a quite nice, if a little bit technical overview of how principal components work for machine learning.

- Most useful for linear-type models
- Can make it harder to interpret predictors
- Watch out for outliers!
    - Transform first (with logs/Box Cox)
    - Plot predictors to identify problems
- For more info see
    - Exploratory Data Analysis
    - Elements of Statistical Learning

PML_2-8 Predicting with Regression

This lecture's about one of the most direct and simple ways to perform machine learning using regression modeling. If you've taken the regression modeling class in the data science specialization, then a lot of this material will be familiar to you. We're just using it in the service of performing prediction.

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, usually used with other ML tools

The key idea here is that we're just going to fit a simple regression model. The idea is to fit a line to a set of data. That line will consist of multiplying a set of coefficients by each of the different predictors. Then we get new predictors or new covariance and we multiply them by the coefficients that we estimated with our prediction model. Then we get a new prediction for a new value. This is useful when that linear model is nearly correct, in other words when the relationship between the variables can be modeled in a linear way – in other words as a function of lines. Then this is a useful way to predict. It's very easy to implement, and it's also quite easy to interpret compared to many machine learning algorithms in the sense that you're fitting a set of lines to a data set and the lines are relatively easy to interpret. It can have poor performance in nonlinear settings so it's usually used in combination with other machine learning algorithms on complicated examples.

Example: Old faithful eruptions

Image Credit/Copyright Wally Pacholka http://www.astropics.com/

We're going to be using data on eruptions of geysers. The geysers have a waiting time in between their different eruptions and there's an amount of time that they actually erupt for.

We can load in very easily a data set that contains some information on eruptions for a particular geyser, Old Faithful in the United States, a famous geyser. To start, load the caret package and load the data for the eruptions. Set the seed so that all the analysis can be reproduced. Then create a training set and a test set just like usual. The training set is created because we're going to be building models only in the training set and then applying them in the test set. Look in the training set and see that there are just two variables which make it a very easy example. There is an eruption time and a waiting time where the waiting time is the time between eruptions and the eruptions is the length of time that the geyser was erupting.

Example: Old faithful eruptions

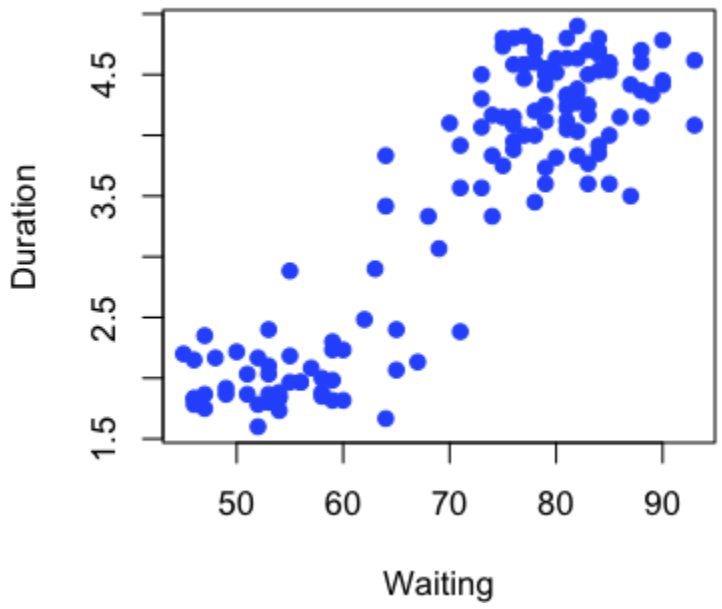
```
library(caret);data(faithful); set.seed(333)
inTrain <- createDataPartition(y=faithful$waiting,
                               p=0.5, list=FALSE)
trainFaith <- faithful[inTrain,]; testFaith <- faithful[-inTrain,]
head(trainFaith)
```

```
   eruptions waiting
6      2.883      55
11     1.833      54
16     2.167      52
19     1.600      52
22     1.750      47
27     1.967      55
```

Eruption duration versus waiting time

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

Make a plot of these two variables and note the waiting time on the x axis and duration time on the y axis. You can see that there's roughly a linear relationship. Imagine drawing a line through these points that predicts relatively well, the duration time from the waiting time.



Fit a linear model

Fit a formula that's a line with the formula for a line that is the eruption duration equal to a constant (an intercept term) plus another constant multiplied by the waiting time plus the error term.

$$ED_i = b_0 + b_1 WT_i + e_i$$

As the graphic above even if a line fit through the middle of the data looks like it's sort of a reasonable approximation to the relationship, it's obviously not. The points don't exactly fall in a line. That's why we allow for some error in our model. The error models, everything that we didn't have, we didn't measure, we didn't understand about the relationship. We use the lm command in R to fit a linear model. **`lm()`** relates the eruptions, the outcome variable that you're trying to predict and the tilde (**`~`**) says we're going to predict it as a function of the predictor variable **`waiting`** (from the waiting variable data in the training data set).

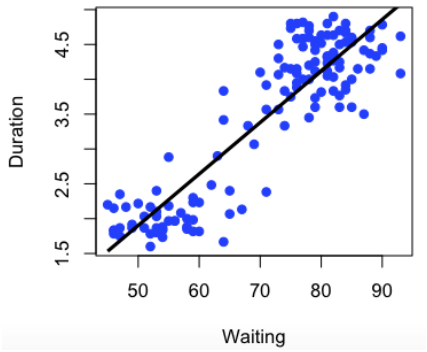
```
lm1 <- lm(eruptions ~ waiting, data=trainFaith)
summary(lm1)
```

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

Residuals:
    Min      1Q  Median      3Q     Max
-1.2699 -0.3479  0.0398  0.3659  1.0502

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.79274    0.22787   -7.87    1e-12 ***
waiting      0.07390    0.00315   23.47   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The model is built using the data from the training set. We then get a summary of the output and the part to look at for the prediction is **summary(lm1)$Estimate**. The estimate **-1.79274** is just the intercept that's the constant b_0 in the formula. The waiting time estimate here is b_1 in the formula. A new prediction for the expected duration is $ED_i = -1.79 + 0.073WT_i$ where WT_i is the new waiting time.

Model fit

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



This is what the model fit looks like by plotting the **train** set, the waiting times versus the eruptions (**trainFaith$waiting,trainFaith$eruptions**). Then plot the fitted values by extracting that from the linear model **nar** object with **lm1$fitted** which will gives the fitted values. Then I plot that versus the predictor variable (**trainFaith$waiting**) used to predict the values.

The graphic here is the waiting time plotted versus duration. The points come from this first plot command and then the black line is plottedt. It comes from the **lines** command that's adding a line with the fitted values. Note that the line is a reasonably good representation of the relationship between the two variables. Obviously the points don't lie exactly on the line, but it's a reasonably good capture of the main data set.

Predict a new value

$$\widehat{ED} = \hat{b}_0 + \hat{b}_1 WT$$

To predict a new variable use the estimated value $\hat{b}_0$ and the estimated value $\hat{b}_1$, usually denote those with little hats above the values. Then just multiply them together using the formula above. Remember we don't have an error term in this formula because we don't know what the error term is for this particular [variable] value. Just use the parts that we can estimate. To get the coefficient values from the linear model object, use the **coef**() command that gives the estimates for those two variables. **Coef(lm1)[1]** gives the intercept ($\hat{b}_0$) and **Coef(lm1)[2]** gives ($\hat{b}_1$). The prediction for eruption duration, given a new waiting time of 80, is 4.119 as the time for the eruption duration.

```
coef(lm1)[1] + coef(lm1)[2]*80  # from coef function
```

```
(Intercept)
      4.119
```

You can actually predict using that **lm1** object so you don't actually have to extract the coefficients and multiply them together Create a new data set that is a data frame that has one new value to predict, say a waiting time equal to 80 (**waiting=80**) then use **predict** and pass it the fitted model (**lm1**) from the training set and the new data set (the **newdata** data frame we created). It gives the prediction for the new value. Note that it matches, so **predict** is using the same formula that you would use to actually calculate out the prediction by hand.

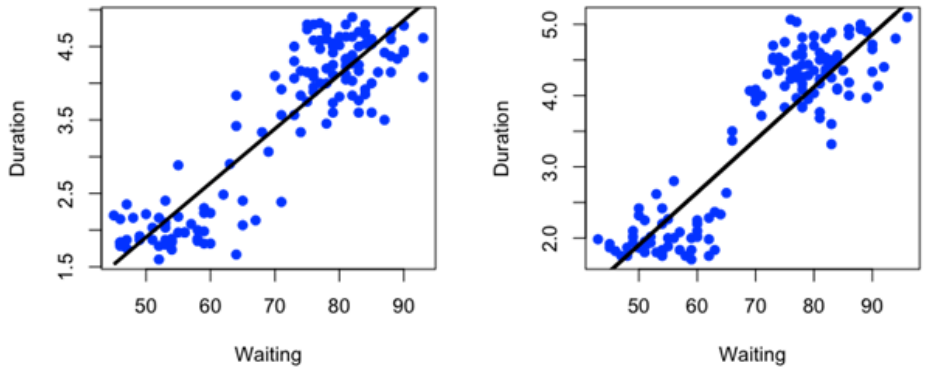
```
newdata <- data.frame(waiting=80)  # newdata is???
predict(lm1,newdata)   # use predict function
```

```
    1
4.119
```

Plot predictions - training and test

Remember that this model is built on the training set and we want to see how it does on the test set. Below note the two plots that show the two separate sets, the training and test sets. On the left is the plot of the training data and on the right is the plot of the test data. Both are plots of the waiting time versus the duration time. Then the model fit (**lm1**) line is added in to both charts with the training data plot displaying a reasonably good model fit. That's to be expected because it is the exact data that we use to build the model. The line plot added to the test data set, the predictions that you get from the model built on the training set. Observe that it doesn't quiet perfectly fit the data anymore like it did in the training set. It's a little tilted underneath [more of the data appears to the left of the line]. But some difference is to be expected because the test set is a slightly different set of data. But as you can see it still captures the overall trend or the overall part of the variation that can be explained by the waiting time.

```
par(mfrow=c(1,2))
plot(trainFaith$waiting,trainFaith$eruptions,pch=19,col="blue",xlab="Waiting",ylab="Duration")
lines(trainFaith$waiting,predict(lm1),lwd=3)
plot(testFaith$waiting,testFaith$eruptions,pch=19,col="blue",xlab="Waiting",ylab="Duration")
lines(testFaith$waiting,predict(lm1,newdata=testFaith),lwd=3)
```



Get training set/test set errors

Next get the training and test set errors. To get the training set error, we get the fitted values, **lm1$fitted**, remember **lm1** was the object actually fitted to the model where the fitted values are the predictions that we get on the training set. Then subtract the actual values of the eruption duration from the prediction on the training set. Square and sum them up then take the square root to calculate the root mean squared error. This measures how close the fitted values are to the real values. The result is 5.752.

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

```
[1] 5.752
```

Next calculate the root mean square error on the test set. Predict again using the **lm1** object that was fitted to the training set. But pass it the new data set, the test data set, **newdata=testFaith** . This predicts values on the test data set.

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

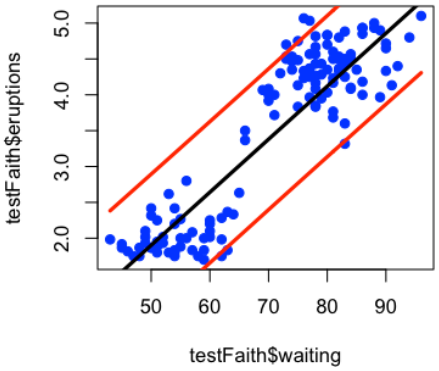
```
[1] 5.839  # more realistic since not part of training
```

Subtract off the actual values of the test data set. Square, sum, and take the square root of the result to find the root mean square error on the test data set as done on the training data set. Since we didn't use the test set at all when we built our algorithm this is a more realistic estimate of the root mean square error that you would get on any new data set compared to the value that we got on the training set. Like always, the test data set error is almost always larger than the training set error because we've used a new set of values that weren't used to calculate the model. That represents the added error, or error and variability, you get when you move to a new data set, [known as] out of sample error.

Prediction intervals

```
pred1 <- predict(lm1,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)   #
Confidence interval plot
```

Another feature that is a nice component of using linear modeling for prediction is to calculate prediction intervals. Again begin by calculating a new set of predictions for the test data set from using our linear model (**lm1**) that was built on the training data set. Also specified is to output a prediction interval, **interval="prediction",** that's just an argument passed to the **predict** function. Then **order()** the values for the test data set and **plot** the test, waiting times versus eruption times.



Add lines to show not only my predictions, the black line shows the predictive values, but also show an interval that is the interval that captures [95] percent of the region where the predicted values are expected to land. Most of the predicted values are expected to land in between the two red lines if the linear model is correct. This also shows a bit about the range of possible values could be predicted – not just a single prediction (black line). These red line bounds can be useful for indicating how well the model is likely to do on new predictions. It shows what the range of likely predictions is.

Same process with caret

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

```
Call:
lm(formula = modFormula, data = data)

Residuals:
    Min      1Q  Median      3Q     Max
-1.2699 -0.3479  0.0398  0.3659  1.0502

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.79274    0.22787   -7.87    1e-12 ***
waiting      0.07390    0.00315   23.47   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.495 on 135 degrees of freedom
```

You can do the same thing in the **caret** package with the **train** function to build the model. The eruption duration (**eruptions**) is the output (outcome) , the waiting time (**waiting**) is the predictor, and they're separated by the tilde (**~**). Then specify which data set to build the model on and for the method use linear modeling, **method="lm"**. Then do a **summary** of that final model fit, **modFit$finalModel,** so the final model is the part of the **modFit** object that was created by the **train** function. That tells you the exact final model that's being used for prediction. Again the results are very similar to the model that was fitted by hand (- 1.79 for the intercept and 0.0739 for the waiting time coefficient).

Notes and further reading

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

Regression modelling can be done with multiple covariates as well. There is a lecture on that. But you can also combine it with all other prediction and machine learning methodology. Again, [linear] regression modelling is a good quick and dirty method for use but it does miss by getting a higher miss-classification error when the relationship isn't necessarily linear.

A lot of prediction is covered with regression modeling in the books mentioned above. They would be a good place to go for more information.

PML_2-9 Predicting with regression, multiple covariates

This lecture is about two things. First it's about predicting with regression and using multiple covariates. Second and more importantly it's about exploring a data set and trying to identify which predictors are the most important to include in our prediction model.

Example: predicting wages

Let's use the wages data to try to predict the wages of a group of men that come from the mid Atlantic region. This data set is available at the ISLR package (see URL) and it comes from the book Introduction to statistical learning.

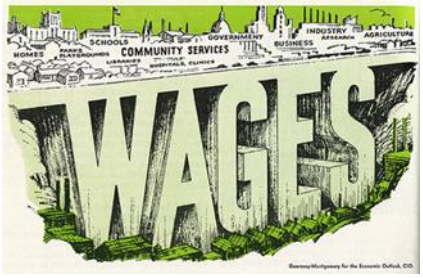


Image Credit http://www.cahs-media.org/the-high-cost-of-low-wages
Data from: ISLR package from the book: Introduction to statistical learning

Example: Wage data

First, load the data set in (the ISLR data set). Next load the ggplot2 package for exploratory analysis. Finally, load the **caret** package for doing prediction. [**names(Wage)** shows the "wage" variable directly and the logarithm of "wage", "logwage"] Next subset the data set for exploration purposes to just the part of the data set that isn't the variable we're trying to predict. Select out the log wage variable [it is highly correlated with the wage variable and would dominate any prediction result]. The summary of the data set shows some features seen before. This data set has only males included in the data set and the region is entirely peopled from the mid Atlantic. [The "wage" variable summary is not displayed in the lecture graphic.]

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

```
 [1] "year"       "age"        "sex"        "maritl"     "race"       "education"
 [7] "region"     "jobclass"   "health"     "health_ins" "logwage"    "wage"
```

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

```
      year           age               sex                    maritl           race
 Min.   :2003   Min.   :18.0   1. Male  :3000   1. Never Married: 648   1. White:2480
 1st Qu.:2004   1st Qu.:33.8   2. Female:   0   2. Married      :2074   2. Black: 293
 Median :2006   Median :42.0                    3. Widowed      :  19   3. Asian: 190
 Mean   :2006   Mean   :42.4                    4. Divorced     : 204   4. Other:  37
 3rd Qu.:2008   3rd Qu.:51.0                    5. Separated    :  55
 Max.   :2009   Max.   :80.0

               education                     region                jobclass               health
 1. < HS Grad      :268   2. Middle Atlantic   :3000   1. Industrial :1544   1. <=Good      : 858
 2. HS Grad        :971   1. New England       :   0   2. Information:1456   2. >=Very Good:2142
 3. Some College   :650   3. East North Central:   0
 4. College Grad   :685   4. West North Central:   0
 5. Advanced Degree:426   5. South Atlantic    :   0
                          6. East South Central:   0
```

Get training/test sets

First subset things into the training and test sets to do a little bit more exploration. Use the Create Data Partition function to subset into a training and test set. Do all exploration on the training set because when we're building models do not use any of the test set. Only apply the model built on the training set once to the test set.

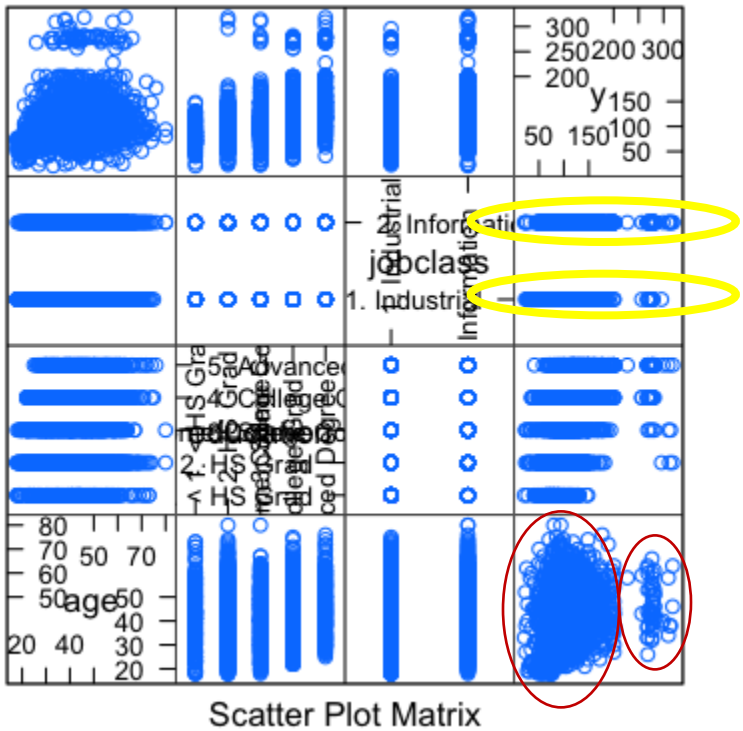
```
inTrain <- createDataPartition(y=Wage$wage,
                               p=0.7, list=FALSE)  # select training set
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
dim(training); dim(testing)
```

```
[1] 898  12
```

Feature plot

One thing to do is create a feature plot. The feature plot for the **Wages** training data set shows how the variables are related to each other. Sometimes this plot is useful and sometimes it isn't. In this particular plot it's a little bit hard to see because everything is so squished together but if you make it on your own by using the **featurePlot** function it's a little bit easier to see. For example, note that for the job class there appears to be two distinct groups and one appears to be higher than the other for the job class in terms of the outcome. Also there is, at least for the age variable and its relationship to the outcome (wage). There seems to be some outlier group, two separate groups here that gives some indication that we might be able to use that variable to predict.

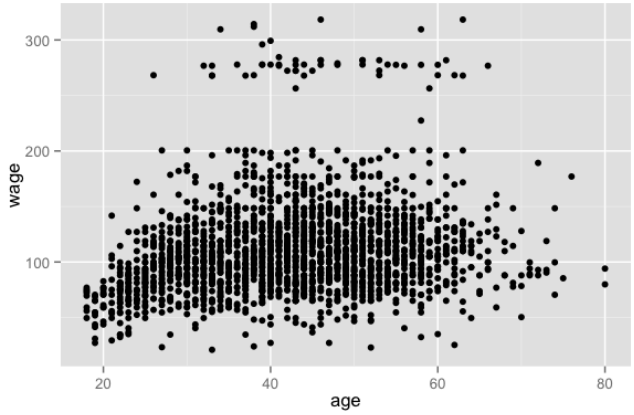
```
featurePlot(x=training[,c("age","education","jobclass")],
            y = training$wage,
            plot="pairs")  # note two age groups (4,4), two job class groups (2,4)
```



Plot age versus wage

Plot the variables versus weight age. See that there appears to be some [arc through the lower cluster] trend which we saw in the previous lectures. But there also is the [upper cluster along the top of the chart]. The idea is that upper set of points might be something that we can predict, but we would have to figure out what variable that is representing that group.

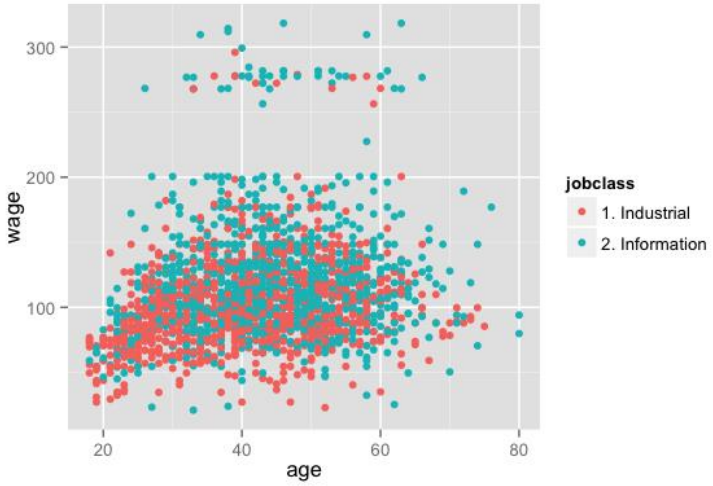
```
qplot(age,wage,data=training)  # note upper cluster data and nonlinear convex up lower
# find which variable represent upper cluster
```



Plot age versus wage colour by jobclass

One way to explore further is to plot one variable (predictor) **age** versus the outcome, **wage**. The color the points by another variable, in this case **jobclass**. And note that it appears that most of these points in the upper cluster are blue instead of pink which means that they come from the information group. The result provides some indication that the **information** variable might be able to predict at least some fraction of the variability that's in that top class up at the top of the plot.

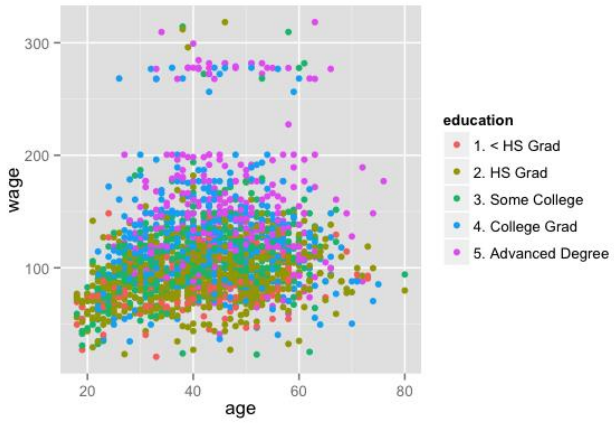
```
qplot(age,wage,colour=jobclass,data=training)  # color data points by jobclass
```



Plot age versus wage colour by education

You can also color it by education in another plot, age versus wage and specifying to color the plot by education. Again use only the training set because the training set is for our development of our model. Note that the advance degree also explains a lot of the variation in the top group. Thus some combination of degree and class of job could explain why the relationship between age and wage isn't just a perfect relationship with all of the data points in one big cloud.

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



Fit a linear model

$$ED_i = b_0 + b_1 age + b_2 I(Jobclass_i = "Information") + \sum_{k=1}^{4} \gamma_k \, I(education_i = levelk)$$

Next fit a linear model with multiple variables in it. The idea fit more than one line. There is an intercept term (b_0) that's the baseline level of wage. Then there is a relationship with the age of the person, and the relationship with job class. One way that is typically done is by fitting an indicator variable. An indicator variable is a variable that's denoted in mathematical notation by $Jobclass_i = "Information"$. It just says if the job class for the i[th] person is equal to information then the job class variable is equal to one ($Jobclass_i = 1$). If the job class for the i[th] person is not equal to information then $Jobclass_i = 0$. The coefficient b_2 is the difference in the wages between the people with job class equal to information versus job class equal to not information when all the other variables in the regression model are fixed. For education it's more complicated because there are multiple education levels. An indicator variable is created for each of the different education levels. In this case it is the sum of four indicator variables, $\sum_{k=1}^{4} \gamma_k \, I(education_i = levelk)$. The variable ($education_i$) is equal to one if the education for person I () is equal to level k and zero otherwise. Note that the coefficient γ_k is the difference in wages for the k[th] education class when age and job class are fixed.

```
#  I(Jobclass=binary (1= Information), I(education=levelk) =binary w four levels (e.g.
[0,0,1,0])
modFit<- train(wage ~ age + jobclass + education,
               method = "lm",data=training)
finMod <- modFit$finalModel
print(modFit)
```

Fit the model as before with the **train** function in the **caret** package. Wage is the outcome and the tilde (**~**) represents the formula one the right (**age + jobclass + education**) that is used to predict the variable on the left (**wage**). Job class and education are both factor variables in R and by default creates the indicator variables ($I(Jobclass_i = "Information")$

and$I(education_i = levelk)$). When it fits the model it takes that into account automatically. Then note again the model is fit on the training set and print out the final model.

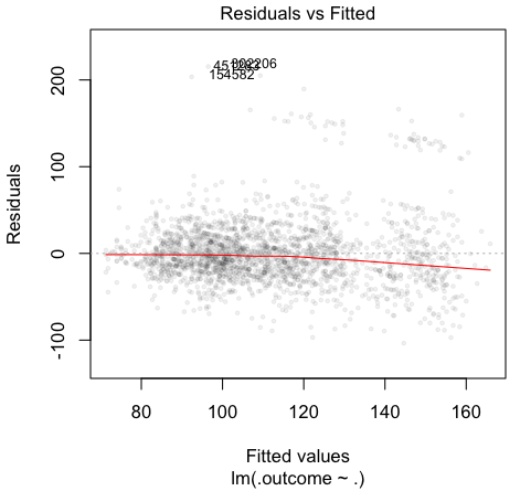
The final model has eleven predictors even though there are only three variable names into the formula. The reason is because variable$I(education_i = levelk)$) received more than one predictor in the data set because of the way the indicator functions are created.
[Recall the 12 variables in **data(Wage)**, ["year" "age" "sex" "maritl" "race" "education" "region" "jobclass" "health" "health_ins" "logwage" "wage"]. **logwage** was removed. Thus the total of predictors is eleven.]

```
Linear Regression

2102 samples
  11 predictors  # note 11 predictors

No pre-processing
Resampling: Bootstrapped (25 reps)

Summary of sample sizes: 2102, 2102, 2102, 2102, 2102, 2102, ...
```

Diagnostics

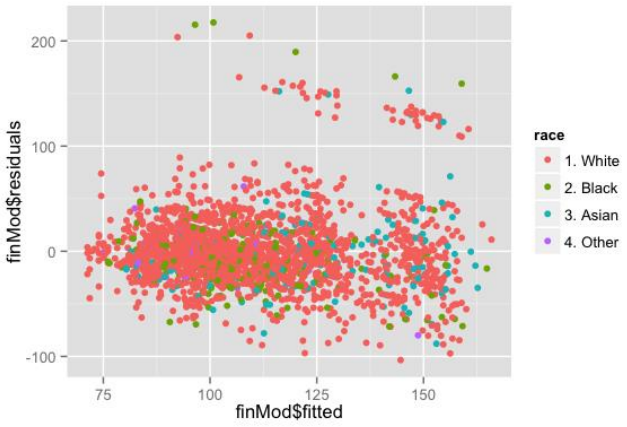
Diagnostic plots assessment is very typical for when building these regression models. The idea here is you can plot the fitted values versus the residuals and the red line shows the prediction from the model on the training set. The residuals show the amount of variation that's left over after you fit your model. What you would like to see is that the red line would be centered at zero on the y- axis because the residuals are the difference between our model prediction and the actual values that we're trying to predict. Also note that there are still a couple of outliers in the upper left of the graphic that have been number labeled in this plot. These might be variables to explore further and see if any other predictors in our data set can be identified that might explain them.

```
plot(finMod,1,pch=19,cex=0.5,col="#00000010")
# plot fitted versus residuals
# Note outliers above, and slight tilt of main residuals cluster
```



Color by variables not used in the model

You can also color by variables not used in the model. For example plot the fitted values from the model versus the residuals. We like to see these data all laying on the y-axis zero line because the plot shows the difference between the fitted values and the real values. The graphic below is plotted with race as the colored parameter. It seems like some of these outliers in the upper right cluster may be explained by the race variable in the data set (most are White). This is another exploratory technique plotting the fitted model versus the residuals then coloring it by different variables to identify potential trends.

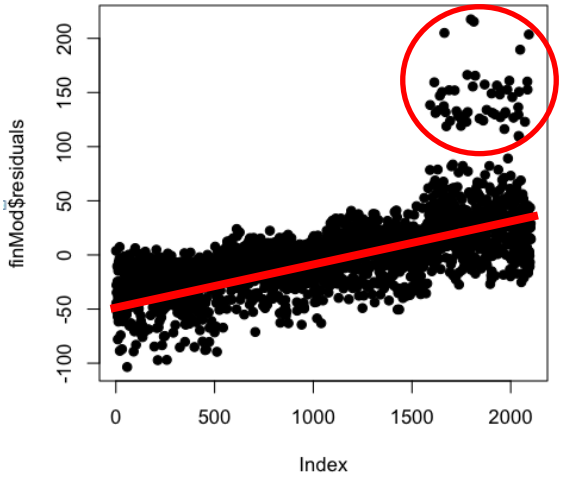



```
qplot(finMod$fitted,finMod$residuals,colour=race,data=training)  # race may explain some of wage disparity in outliers above.
```

Plot by index

```
plot(finMod$residuals,pch=19)  # residuals by row in data set, trend implies some other vactor involved in data collection, time, age, etc.  Also no outlier cluster.
```

Another really useful investigation is plotting the fitted residuals versus the index. What do I mean by the index? The data set comes in a series of rows that come in a particular order. The index is just which row of the data set with an observation and the y-axis in the graphic is the residuals. Note that all the high residuals seem to be happening at the right end of the highest row numbers. Note also see a trend with respect to row numbers for the majority of the observations. Whenever you can see a trend cluster or an outlier cluster with respect to the row numbers, it suggests that there's a variable missing from the model because by plotting the residuals that shows the difference between the true values and the fitted. There shouldn't be any relationship to the order in which the variables appear in your data set unless there's a relationship with respect to time, or age, or some other continuous variable that the rows are ordered by.



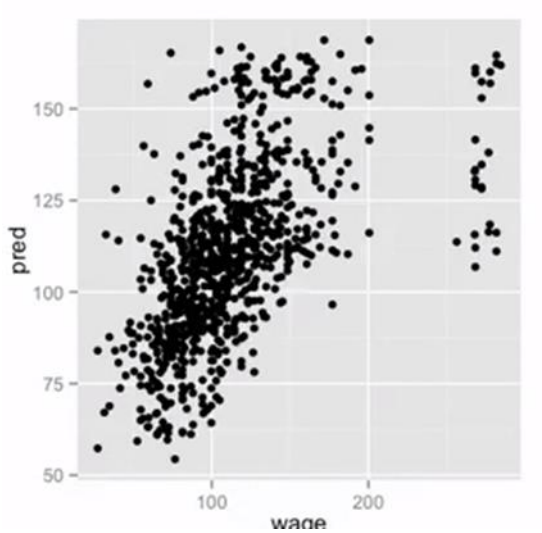
Predicted versus truth in test set

```
pred <- predict(modFit, testing)
qplot(wage,pred,colour=year,data=testing)  # ideal is line with slope = 1.
# IMPORTANT - this is a postmortem perspective.  Cannot go back and use test set data to rebuild predictors!
```

Another useful investigation is plotting the wage (outcome) variable. The graphic shows the **wage** variable in the *test set* versus the *predicted* values in the *test set*. Ideally these two would be very close to each other and you'd have essentially a straight line on a line where wage was exactly equal to predictions. Of course that isn't how it always works out. In the test set you can explore and try to identify trends that you might have missed. For example in the graphic with year as a parameter when the data was collected in the test set (blue shaded colors of the data points). This explores how the model might have broken down. Do keep in mind is that with this exploration in the test set you can't then go back and re-update the model in the training set because that would be using the test set to rebuild the predictors. This is more like a post-mortem of the analysis or a way to try to determine whether the analysis worked or not.

If you want to use all covariates

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



If you want all of the covariants in your model building, in the training function pass it an outcome and then tilde and a dot instead of putting a set of variables separated by plus signs (**`wage ~ .`**). It says predict with all of the variables in the data set. **`modFitAll`** is a model fit with all of the variables and so this is the wage variable and the predictions here and so you can actually see that it does a little bit better when you include all of the variables in the data set.

Use **`~ .`** by default if you don't want to try to do some sort of model selection in advance.

Notes and further reading

Linear regression is often useful in combination with other models. It's a quite a simple model in the sense that it always fits lines through the data and it can capture a lot of variability if the relationship between the predictors and the outcome is linear. If it's not linear, then linear regression can often miss things and it can be better blended with other models. Model blending is covered later in the class.

Exploratory data analysis can be very useful with regression models because plots made with residuals and so forth colored by different features helps to identify the patterns in the data set. There's more information in the three books below, if you want to find a lot more about regression modeling for prediction.

- Linear modeling is often useful in combination with other models
  Other models better at capturing nonlinearities. *"Exploratory data analysis can be very useful with regression models because, like, the plots we made with residuals and so forth colored by different features, you can try to identify the patterns in the data set."*
- Elements of statistical learning
- Modern applied statistics with S
- Introduction to statistical learning