Week 2: Caret package

Index:

Week 2.1: Caret package

The caret package is a very useful front-end package that wraps around a lot of prediction algorithms and tools in R.
http://topepo.github.io/caret/index.html

Caret functionalities

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

Machine learning algorithms in R

Many, for example:

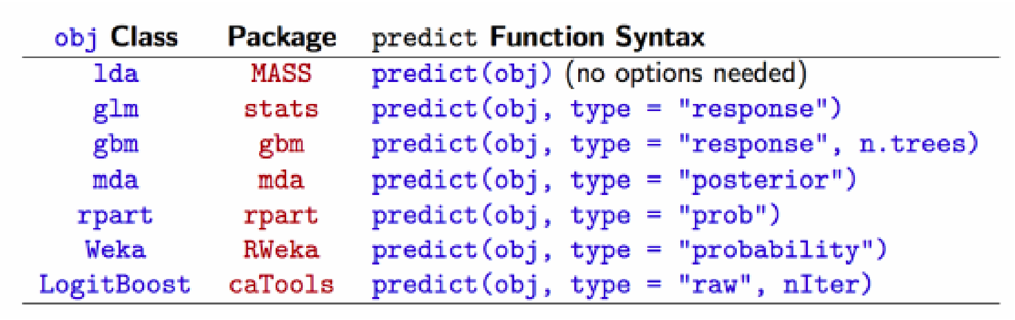
- Linear discriminant analysis
- Regression
- Naive Bayes

- Support vector machines
- Classification and regression trees
- Random forests
- Boosting
- ...

All these algorithms have been built by a variety of different developers, all coming from different backgrounds, so the interfaces for each of these prediction algorithms is slightly different.

Why caret?

For example, consider these classes of different prediction algorithms:



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)

For each of these different algorithms you could create an R object ("obj"), which will have a different class for each algorithm used. When we try to use the predict function on each object we need to add slighlty different parameters to the function for each different algorithm.

The caret package provides a **unifying framework** that allows prediction using just one function and without having to specify all the options that would be required otherwise.

Example of using the caret package:

Spam example - Data splitting

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

```
[1] 3451   58
```

The `spam` dataset from the `kernlab` library gets split into a training (75%) and a test (25%) set using the `createDataPartition` command. The data is actually subset using the created `inTrain` object into `training` and `testing` datasets.

Spam example - Fit a model

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

- In this step we used the `train` command from `caret` package. We want to predict spam `type` and we used `~.` to use all the other variables in the dataframe to predict `type` target variable.
- `data = training` specifies which training dataset we want to buil the model on.
- `method = 'glm'` is used to choose the method to use (GLM in this example)
- the `train` caret function can do a bunch of different ways of testing whether this model will work well and use it to select the best model (in this example, resampling/bootstrapping was used)

Spam example - Final model

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

```

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
         labs         telnet         num857           data         num415
```

- Once the model has been fitted, we can look at it using the component `modelFit$finalModel` of the `modelFit` object. It will tell you what are the actual fitted values that you got for that GLM model.

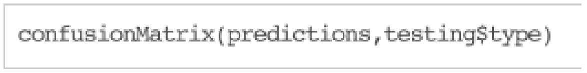
Spam example - Prediction:

```
predictions <- predict(modelFit,newdata=testing)
predictions
```

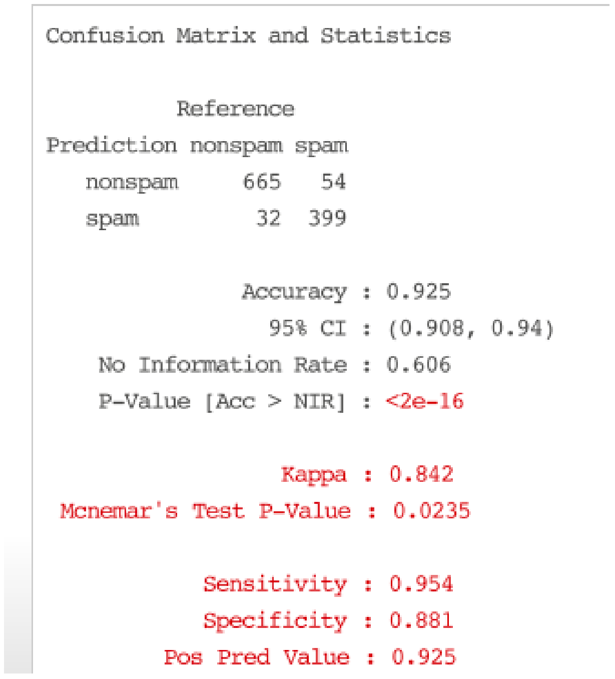
```
  [1] spam    spam    spam    nonspam nonspam nonspam spam    spam    spam    spam    spam   
 [12] spam    spam    spam    spam    spam    spam    spam    nonspam spam    spam    spam   
 [23] nonspam spam    nonspam nonspam spam    spam    spam    spam    spam    spam    spam   
 [34] spam    spam    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   
[177] spam    spam    spam    spam    spam    spam    spam    spam    spam    spam    spam   
```

- You can then predict on new samples using the caret `predict` command and by passing it the `modelFit` object we created with the `train` function in caret.
- In this case the new data is the `testing` data. The command will return a set of predictions that correspond to a set of responses (e.g. spam/nonspam), which can then be used to try and evaluate whether your model fit works very well or not.

Spam example - Confusion Matrix:



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



```
Confusion Matrix and Statistics

          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
            Specificity : 0.881
         Pos Pred Value : 0.925
```

- You need to pass the `predictions` into caret's `confusionMatrix` function
- The output is the confusion matrix and a bunch of other statistics and other useful information to evaluate the model's fit.

Useful caret tutorials

https://www.r-project.org/conferences/useR-2013/Tutorials/kuhn/user_caret_2up.pdf

https://cran.r-project.org/web/packages/caret/vignettes/caret.html

https://www.jstatsoft.org/article/view/v028i05/v28i05.pdf

Week 2.1: Data slicing

You might use data slicing for either:

- Building your training and testing sets at the beginning of your prediction function creation.
- Performing cross-validation or bootstrapping within your training set in order to evaluate your models.

Spam example - Data splitting

- We can use `createDataPartition` to separate the datasets into training and test sets right at the beggining, before the training step.
- We need to specify on which outcome we want to split on, in this case `y=spam$type`, and the proportion of the sample that goes into the training set with `p=0.75` (75% of sample).
- Following those rules, the `inTrain` variable gets assigned an indicator function which I can use to subset out the training and the testing set from the original `spam` dataset.

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

```
[1] 3451   58
```

Spam example - K-fold

- This is to carry out cross validation, where you split your training set into a bunch of smaller sets (K-folds) that you will use to do cross-validation.
- To do this you can use the caret's `createFolds` function, passing the outcome you want to split on (`y = spam$type`) and the number of folds you want to create (`k=10`). Setting `list=True` is used to return each set of indices that corresponds to a particular fold as a set or as a list. We also indicate to return the training set with `returnTrain = True`

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

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

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

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

- The command `folds[[1]][1:10]` returns the first 10 elements of the first fold as a list. You can see they correspond to the first ten elements of the sample --> the split up occurs in the order of the original dataset

- The sample has been split into train:test 75%:25% (I think this should be 9:1), so if we use `returnTrain = False` and return the test set, we can see it is much smaller:

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

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

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

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

Spam example - Resampling

- You can use the caret's function `createResample` to perform resampling/bootstrapping.
- You need to specify how many times you want to resample the data (`times = 10`) and whether you would like a list/vector/matrix out (to choose a list: `list = True`)

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

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

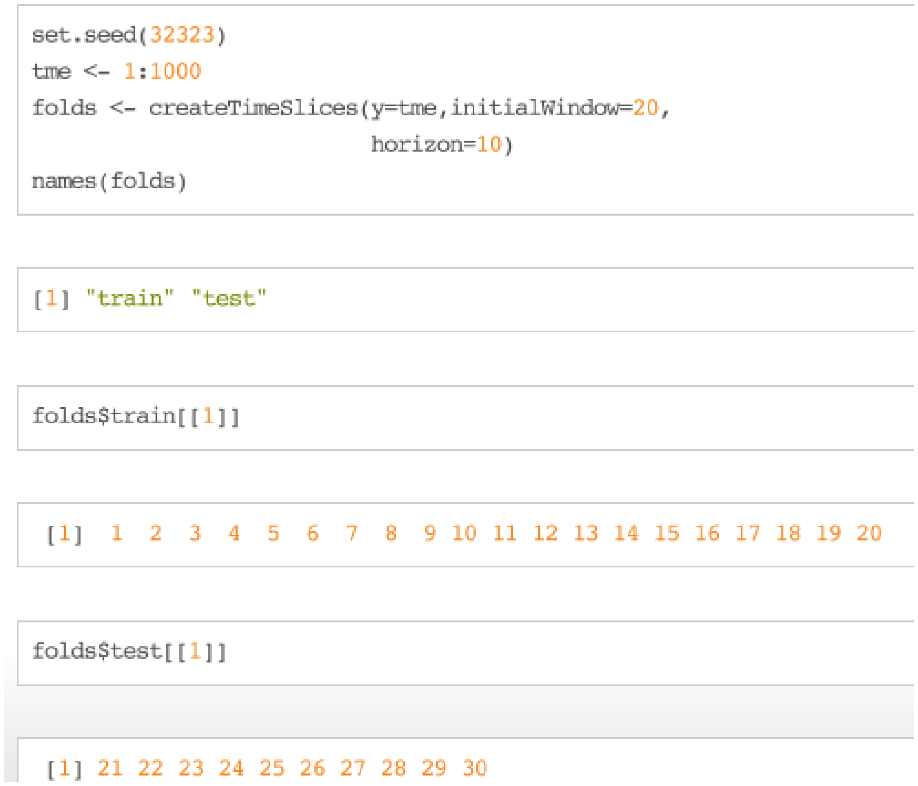
```
folds[[1]][1:10]
```

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

- Since we are doing resampling, you might get some of the same values back within the same subsampe (e.g. samples '3' and '5' repeated in the first fold)

Spam example - Time slices

- You use the caret's command `createTimeSlices` to analyse data that you use for forecasting to create time slices that include continuous values in time.
- Use parameter `initalWindow` to specify time windows (in the example containing about 20 samples).
- Parameter `horizon` is used to specify the number of samples you want to predict occurring after the predictive window



```
set.seed(32323)
tme <- 1:1000
folds <- createTimeSlices(y=tme,initialWindow=20,
                          horizon=10)
names(folds)
```

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

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

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

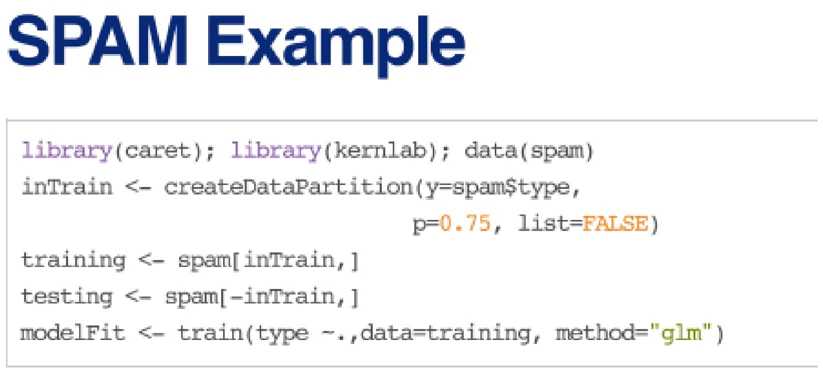
```
folds$test[[1]]
```

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

- The time variable order of the sample is used to predict in this way

Week 2.2: Training options

Using the caret package to train model, you have a few training control options available.



SPAM Example

```
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")
```

- The caret `train` function has been used above using all the default parameters (only the model, input `data` and `method` have been specified)
- You can decide instead, to use a large set of options for training, e.g.:

- `preProcess` parameter to **set preprocessing options**
- `weights` parameter is used to **upweight or downweight certain observations** (useful, for example, if you have a very unbalanced training set where you have many more examples of one type than another)
- `metric` is to set the metric/error we want to maximise/minimise (for factor/categorical variables the default metric that the model is trying to maximise is `"Accuracy"`, while for continuous variables it is trying to minimise the `"RMSE"` or Root Mean Squared Error)
- `trControl` is used to set a large number of other control parameters. You do this by passing a call to the function `trainControl()`

```
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)
NULL
```

Metric options

Continuous outcomes:

- **RMSE** (Root Mean Squared Error) - caret default for continuous variables
- **Rsquared** (R^2 from regression models), it measures the linear agreement between the variable that you are predicting and the variables you are predicting with, which is very useful when you are using linear regression (not so useful for non-linear algorithms, such as random forests etc)

Categorical outcomes:

- **Accuracy** = Fraction of correctly predicted samples on total number of predicted samples - caret default for categorical variables
- **Kappa** = a measure of concordance, more complicated than accuracy

caret `trainControl`

The trainControl argument allows you to be much more precise about the way you train your model

```
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)
NULL
```

- `method` to specify the method to use for resampling your data (e.g. bootstrapping or cross-validation)
- `number` to specify the number of times to perform resampling
- `repeats` is to carry out repeated cross-validation
- `p` is to specify the size of the training set
- `initialWindow` to specify the size of the training set for time series data (e.g. number of training time points)
- `horizon` to specify the size of the testing set for time series data (e.g. number of testing time points)
- `savePredictions` is to return the predictions itself from each iteration when it is building the model
- `summaryFunction` different form `defaultSummary` can be specified
- `preProcOptions` is to set pre-processing options
- set `predictionBounds`
- set `seeds` for all the different resampling layers
- `allowParallel` parameter is useful if you are going to parallelise your computation across multiple cores

`trainControl` resampling

Important parameters for resampling:

- `method`:
    - `boot` = bootstrapping
    - `boot632` = bootstrapping with adjustements (to reduce some of the bias due to the fact that multiple samples are repeatedly resampled)
    - `cv` = cross-validation
    - `repeatedcv` = repeated cross-validation (to do sub cross validation with different random draws)
    - `LOOCV` = leave one out cross-validation
- `number`:
    - for bootstrap and cross-validation

    - number of subsamples to take
- `repeats`:
    - number of times to repeat subsampling (e.g. when doing repeated cross-validation)
    - if big, it can slow things down!

Setting the seed

An important component of training these models is setting the seed, since most of these procedures rely on random resampling of the data and if you rerun the protocol/code over again you will get a slightly different answer because there was a random draw that was created when you were doing your cross-validation. If you set a **random number seed**, you will ensure that the same random numbers get generated each time (it always generates the same sequence of pseudo-random numbers).

- It is often useful to set an overall seed (for the entire procedure)
- You can also set a seed for each resample
- Seeding each resample is useful for parallel fits
- Important when you want to share your code and be sure other people will get the same results
- It can be done by using the `set.seed()` command

Example:

```
set.seed(1235)
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
```

Week 2.3: Plotting predictors

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

Example: predicting wages

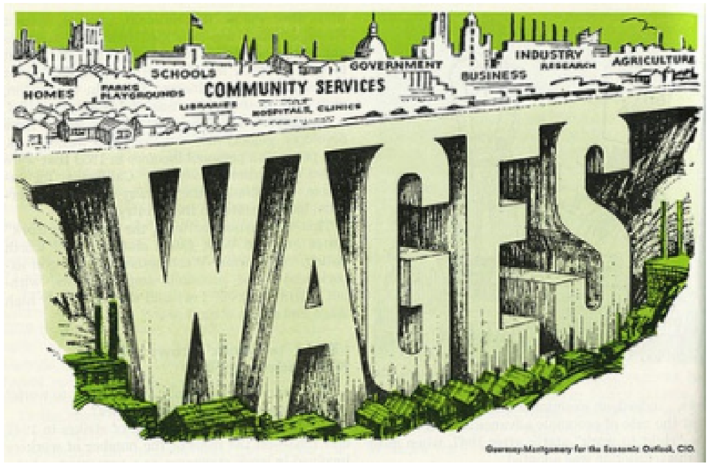
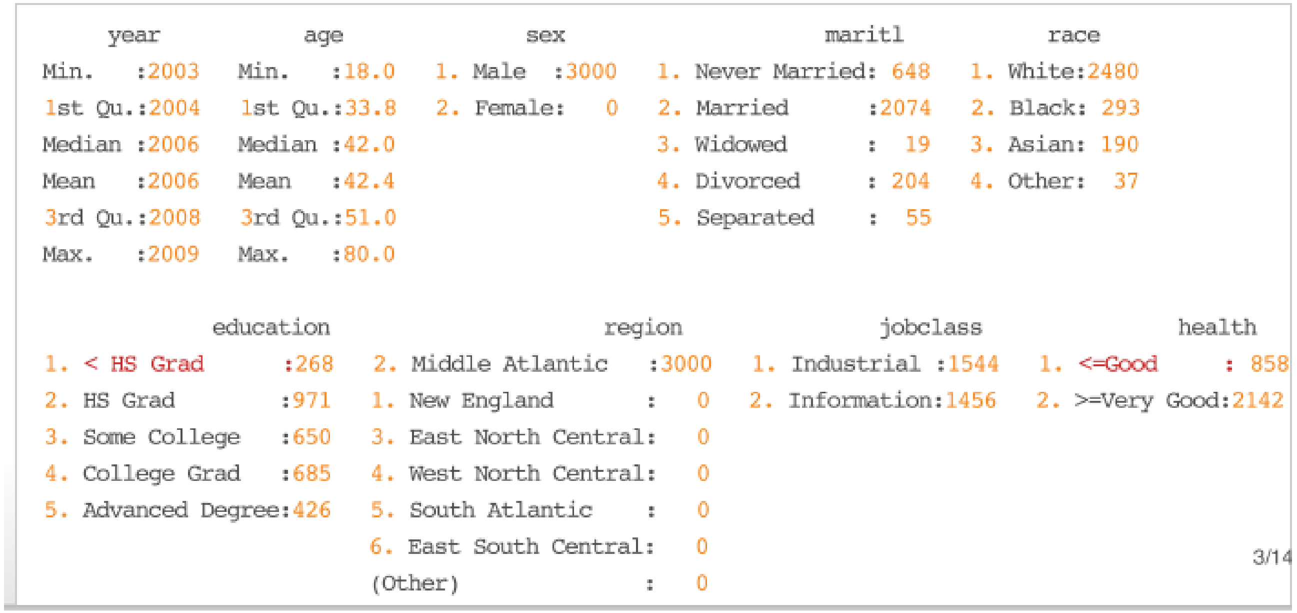


Image Credit http://www.cahs-media.org/the-high-cost-of-low-wages

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

```r
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
                           (Other)              :    0
```

3/14

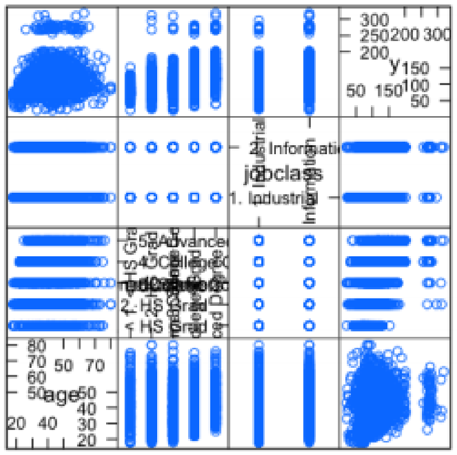
- Even before we do the data exploration, we are going to set aside the testing set and we are not going to use it for anything until we look at the data at the end of the building model experience and apply it just one time:

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

```
[1] 898  12
```

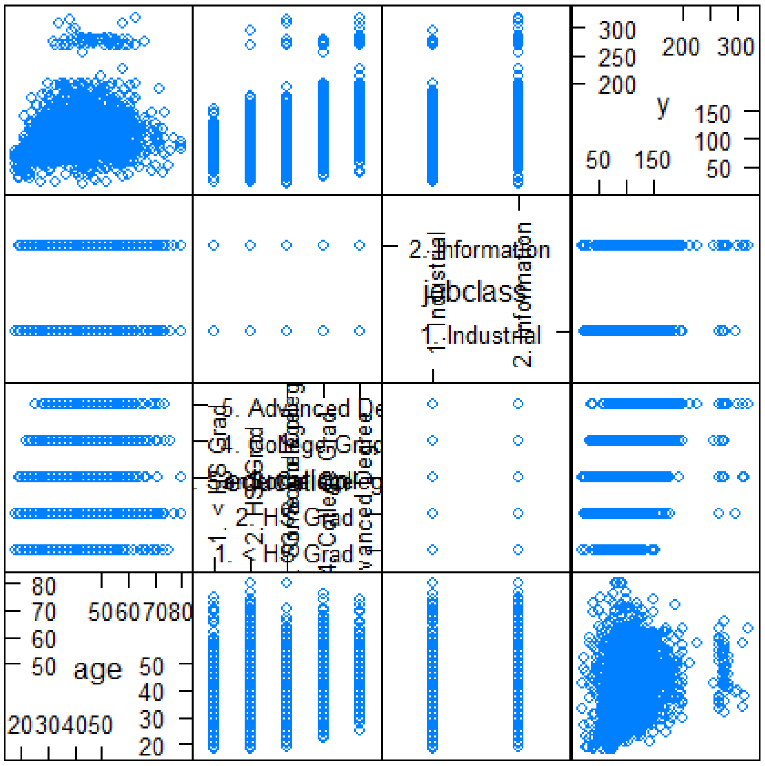
- We are going to do all our plotting in the training set.
- We can use a **feature plot** from caret package (`featurePlot` function), which will **plot all features against each other**:

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



Scatter Plot Matrix

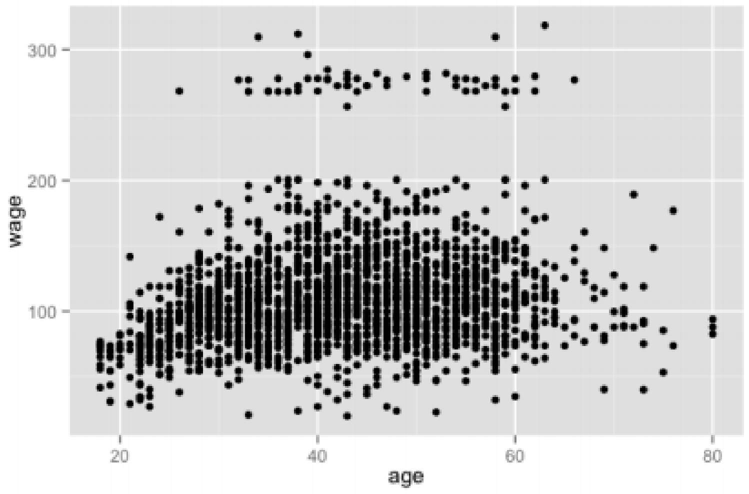
...trying to make it a bit clearer:



Scatter Plot Matrix

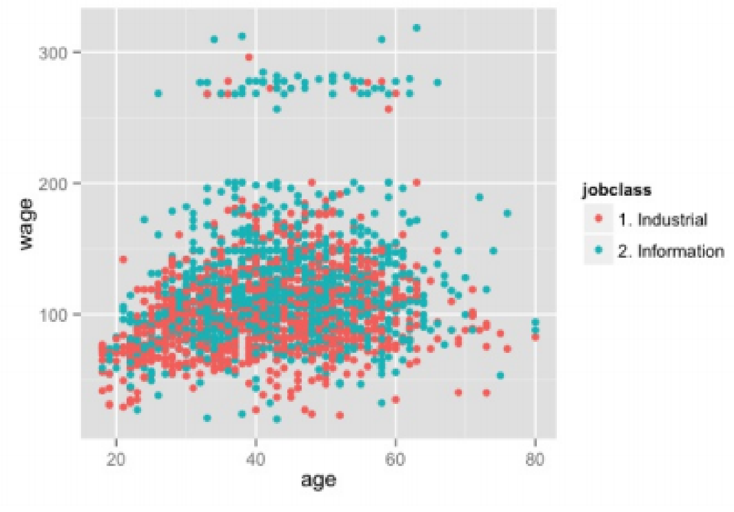
- You are particularly interested at all the plots of the select features against the y-variable (first upper row) and look for any variable that seems to show a relationship with the y variable (e.g. there seems to be a trend between education and salary)
- You can also use the `qplot` function from the `ggplot2` library (or just the `plot` function in base R):

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



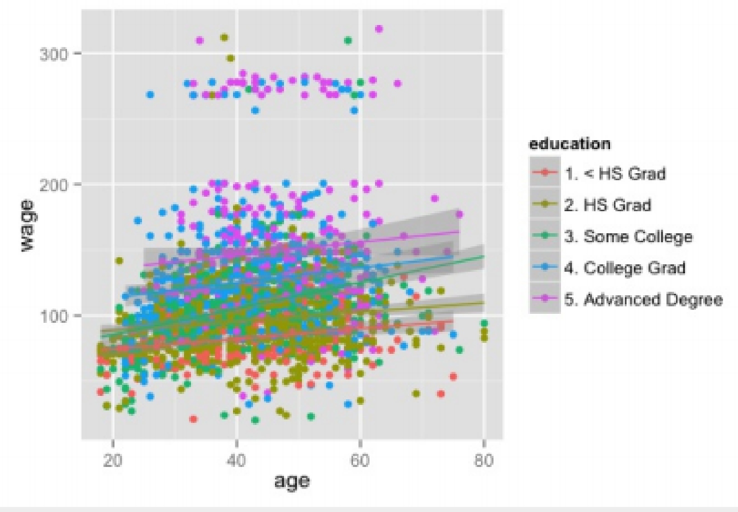
- There seems to be some kind of trend between Wage and age. There is also a chunk of data in the upper portion that relates differently from the rest of the data --> you can then try and color that plot by different variables:

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



- We can now see most people with the highest wages in the top part of the plot come from an 'information' type of job (as opposed to the 'industrial' job).
- In these ways we can identify variables that might be important in the model because they show variation in the data.
- You can also add **regression smoothers**, so for every specified class type (for example 'education') a regression model is fitted:

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



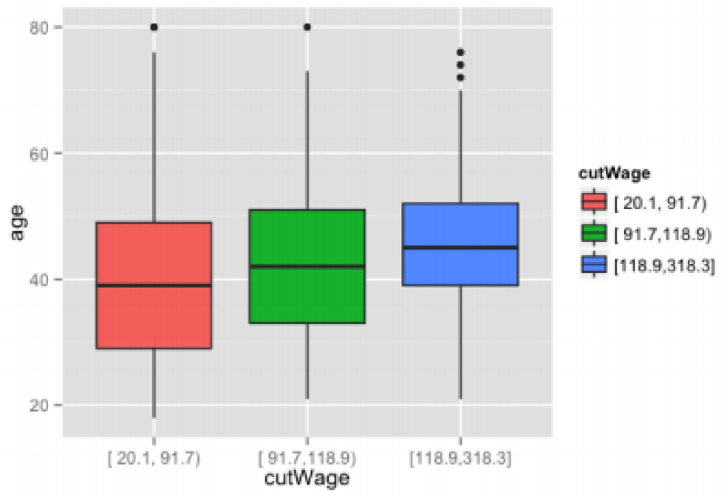
Another thing we can do is to break continuous/numeric variables into categories which might different relationships with the other variables (for example the "wage" variable into three levels). You can do this using the `cut2` function from the `Hmisc` package, which breaks the data into factors based on quantile groups:

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

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

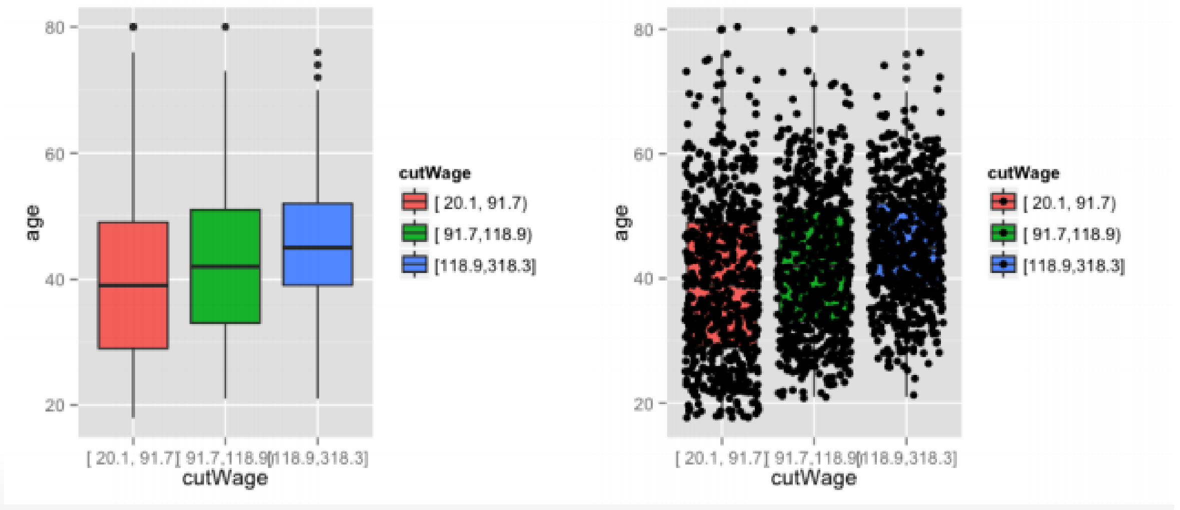
- You can use the different levels of the outcome variable to make other plots, e.g. boxplots of wage category versus age:

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



- you might want to add on top of the boxplots the data points as well to visualise better the data distributions:

```
p2 <- qplot(cutWage,age, data=training,fill=cutWage,
      geom=c("boxplot","jitter"))
grid.arrange(p1,p2,ncol=2)
```



- you can also use the factorised variable to create tables of data:

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

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

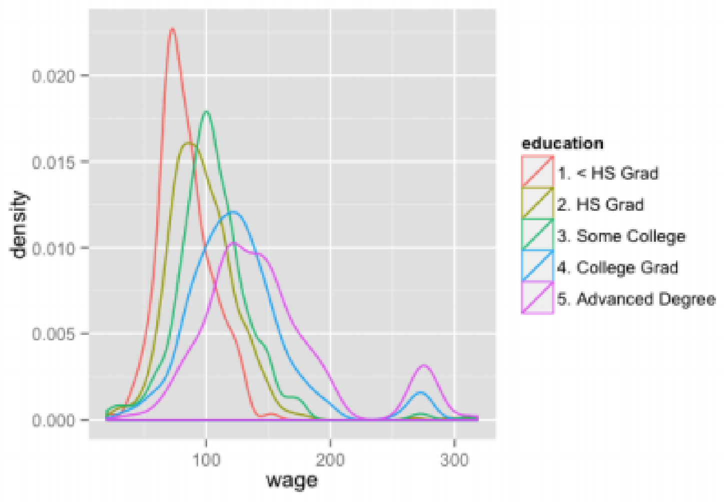
```
prop.table(t1,1)
```

```
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
```

- it is visible from the tables that there are more industrial jobs in the low wage class, while there are more information jobs in the high wage class.
- `prop.table` gets the proportions in the table (1 to have proportions by row, 2 by column)

- You can also use **density plots** to plot the values of continuous predictors (e.g. wage by education class):

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



- This plot shows where the bulk of the data is.
- It is also easier with density plots to lay different groups

Notes and further reading

- 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** (e.g. if all a predictor's values tend to be one value in one of the outcome class and another value in the other outcome class, this is a sign that is a very good predictor, able to differentiate between outcomes' classes)
    - **Outliers** (they might suggest that some variables are missing)
    - **Group of points not explained by a predictor**
    - **Skewed variables** (you might want to transform them and make them more normally distributed if you are using algorithms such as linear regression)

http://topepo.github.io/caret/visualizations.html

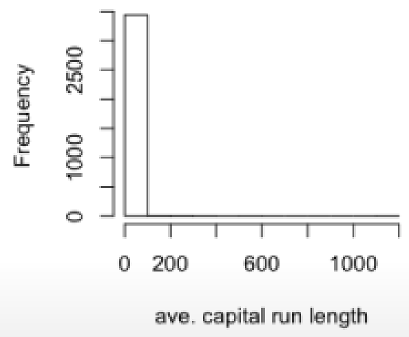
Week 2.4: Preprocessing

We will talk about **pre-processing predictor variables**. After plotting you might have noticed that some predictors look a bit strange (e.g. their distribution is not normal) and they might need to be transformed so they can be used in the best way by the ML algorithms. This is particularly true when we are using **model-based algorithms** (rather than non-parameetric approaches), such as **linear discriminant analysis**, **naive Bayes**, **linear regression**, etc ...

Why preprocess?

- When you decide to pre-process data you only look at the training set.

```
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")
```



- In the spam data we plotted the number of capital letters we see in a row (`capitalAve` variable): almost all the values for this variable are very small, although few are much larger --> example of a **variable that is very skewed** and it is very hard to use it in model-based predictors, so you want to pre-process it.

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

```
[1] 4.709
```

```
sd(training$capitalAve)
```

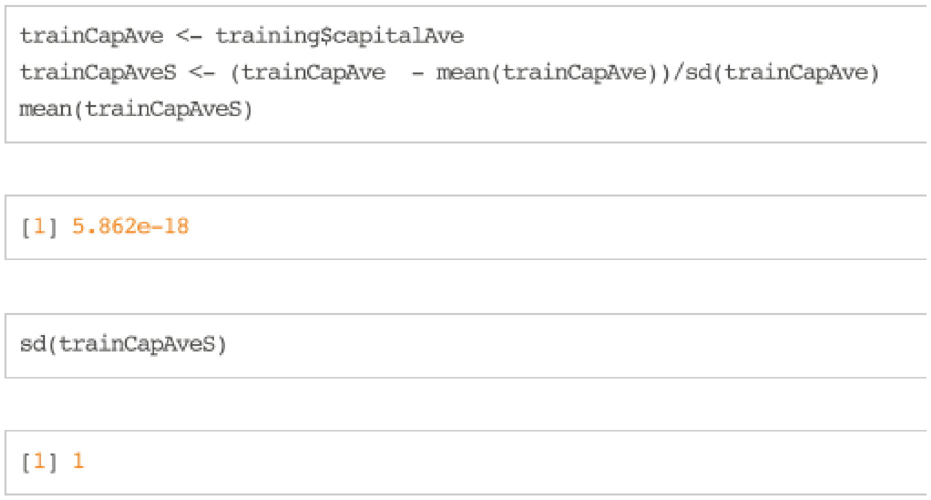
```
[1] 25.48
```

- The standard deviation is very high

Standardising

- You want to standardise the skewed variable, by subtracting by each of its values its mean and then dividing the result by the var's standard deviation. In this way the new **standardised variable** will have

mean = 0 and **std dev = 1**.



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

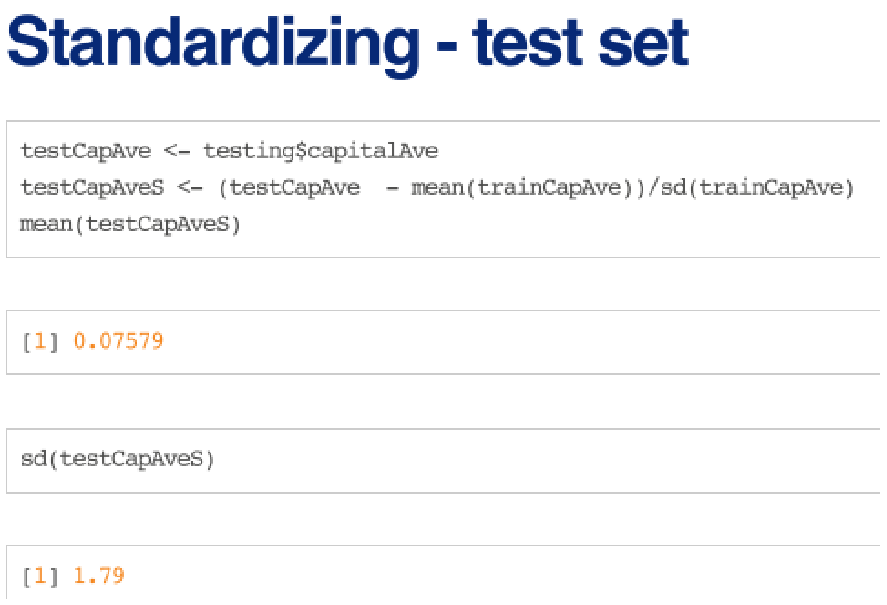
```
[1] 5.862e-18
```

```
sd(trainCapAveS)
```

```
[1] 1
```

Standardising - test set

- When we apply a prediction algorithm to the **test set**, we can **only use parameters that have been estimated in the training set**. Thus, when we apply the same standardisation to the test set, we have to use the **mean and the standard deviation paremeters obtained from the training set**.
- Because of this, when you apply the standardisation in the test set, for the new standardised variable the mean will not be exactly 0 nor the std dev exactly 1.



Standardizing - test set

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

```
[1] 0.07579
```

```
sd(testCapAveS)
```

```
[1] 1.79
```

Standardising: caret's `preProcess` function

- In the following code, all variables (except the outcome variable) of the training set are passed to the `preProcess` function.
- The `method=c("center", "scale")` subtracts the mean from and divide by std dev each value (using the distribution parameters of each variable):

```
preObj <- preProcess(training[,-58],method=c("center","scale"))
trainCapAveS <- predict(preObj,training[,-58])$capitalAve
mean(trainCapAveS)
```

```
[1] 5.862e-18
```

```
sd(trainCapAveS)
```

```
[1] 1
```

- You can also use the object created using the preProcess function on the training set (preObj) to apply the same standardisation on the test set:

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

```
[1] 0.07579
```

```
sd(testCapAveS)
```

```
[1] 1.79
```

- You can also pass the `preProcess` command directly into the `train` function in caret as an argument
- In the example, all the predictors in the training set get centred ans scaled:

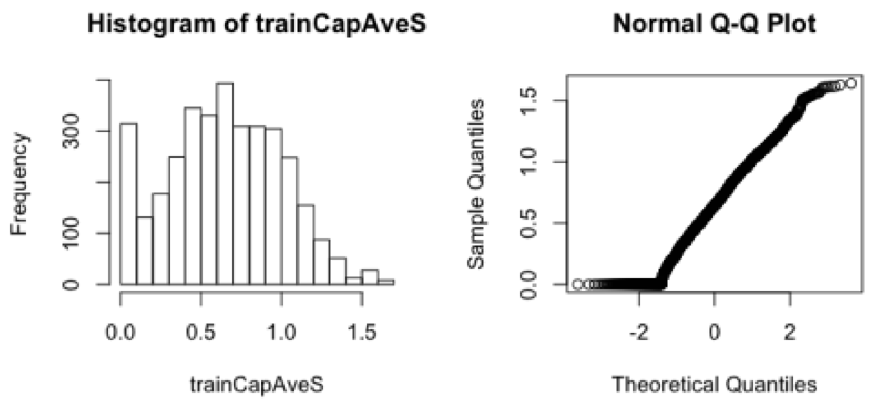
```
set.seed(32343)
modelFit <- train(type ~.,data=training,
                  preProcess=c("center","scale"),method="glm")
modelFit
```

```
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
```

Standardising: Box-Cox tranforms

- You can also carry out other kinds of tranformations apart from centering and scaling
- One example is **Box-Cox transforms**, a set of transformations that **take continuous** data and try to **make them look like normal data** by estimating a specific set of parameters using **maximum likelihood**.
- This transformation can be carried out on all the training variables using the `preProcess` caret function:

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



- After the `BoxCox` transformation, the variable (`capitalAve`) distribution looks more normal, although there are still issues (e.g. a stack set of values at 0 on the histogram and a long tail in the left bottom

part of the normal Q-Q plot).
- The BoxCox does not take care of repeated 0 values

Standardising: Imputing data

- It is very common to have **missing data** and it is a problem because they make often fail the prediciton algorithms (as in most cases are built not to handle missing data).
- You can impute missing data using the **k-nearest neighbors' imputation**:

```
set.seed(13343)

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

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

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

- We set the seed because the imputation algorithm is randomised and we want to get reproducible results
- the `rbinom` function is used to randomly generate a bunch of NA values to add to the `capAve` variable.
- In order to handle the missing value, you can use the `preProcess` caret function and choose `method = 'knnImpute'`
- KNN imputation finds the k-nearest data vectors that look most like the data vector with the missing value, average among them the value of the variable that is missing and impute them at the NA position.
- You use then the `predict` command to impute the new values for the missing values using the KNN imputation algorithm
- All the values for the variable (included the imputed ones) finally get standardised in the code ((value-mean)/(std dev))

```
quantile(capAve - capAveTruth)
```

```
       0%        25%        50%        75%       100% 
-1.1324388 -0.0030842 -0.0015074 -0.0007467  0.2155789 
```

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

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

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

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

- We can then look at the difference between the variable's values that were imputed (contained in `capAve`) and those that were not imputed (contained in `capAveTruth`), to see how close they are to each other.
- You can see from the results of the first row (`quantile(capAve - capAveTruth)`) that most of the differences between imputed and non-imputed values are very close to 0, so the imputation worked relatively well
- The second row (`quantile((capAve - capAveTruth)[selectNA])`) shows that also for only the imputed values most of the difference is close to 0, although the data is a bit more variable
- The third row (`quantile((capAve - capAveTruth)[!selectNA])`) shows that the non-imputed values are also closer to each other than the imputed ones.

Notes and further reading

- Training and test sets must be processed in the same way (the caret package deals with this a lot under the hood: if you train a dataset using the `preProcess` function built into the `train` function, it will apply the same `preProcess` function to the test set. In this way all the pre-processing is handled in the correct way for you). Anything you do to the training set will create a set of parameters that need to be applied to the test set as well.
- Test transformations will likely be imperfect (especially if the test/training sets were collected at different times)
- All the transformations described in this Week, except imputation, are based on continuous variables. Careful when transforming factor variables! Most ML algorithms are built to deal either with binary predictors that have not been pre-processed or continuous predictors (for some of the latter, some pre-processing is expected if they need to look more normal).

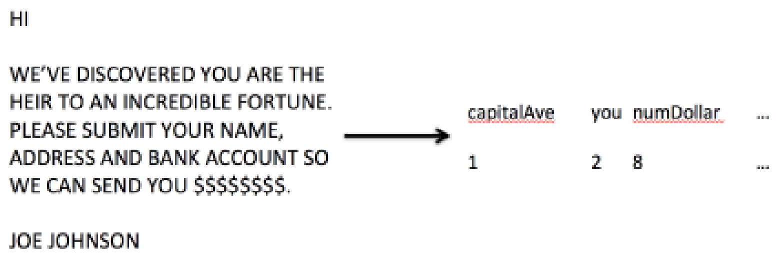
https://topepo.github.io/caret/pre-processing.html

Week 2.6: Covariate creation

Covariates are sometimes called **predictors** or **features**. They are the variables you will actually include in your model and using by combining them to predict the target outcome.

Two levels of covariate creation

1) From **raw data** to **covariates**: the raw data is transformed into predictors that can be actually used (e.g. when the raw data takes the form of an image, text file or a website). The information in the raw data needs to have been summarised in some useful form in order to be used in the model (i.e. as a quantitative or qualitative variable). We take the raw data and turn it into covariates which describe the data as much as possible while giving some compression and making it easier to fit standard ML algorithms. This step usually involves a lot of thinking about the structure of the data that you have and how to extract the most useful information in the fewest number of variables so to capture everything you want.

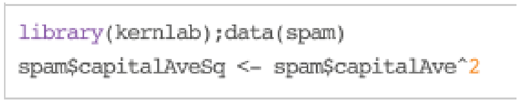

HI

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
...

2) Transforming **tidy covariates**: the variables created in step 1 are transformed into sort of more useful variables.



```
library(kernlab);data(spam)
spam$capitalAveSq <- spam$capitalAve^2
```

Level 1: Raw data -> covariates

- It depends heavily on application
- The balancing act is **summarisation** vs **information loss**. The best features are those that captures only the relevant information and throw out all the rest.
- 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 types of images, position of elements, colours, videos (A/B testing, also called randomised control trial or RCT in stats. It consists in showing different versions of a

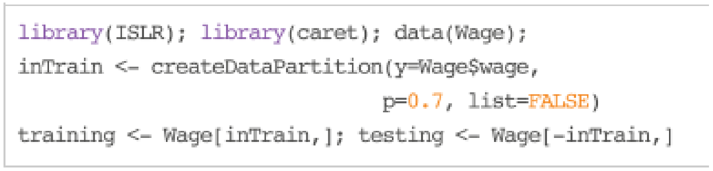
webiste with different values of different features and predicting which ones will introduce more clicks or get more people to buy products)
    - People: height, weight, hair colour, sex, country of origin
- The more knowledge of the system you have, the better job you will do (it is a good idea to have a clear understanding of why this set of data is useful for predicting the outcome you care about)
- When in doubt, err on the side of more features (so you lose less information)
- It can be automated, but use caution (sometimes a particular feature will be very useful in the training set you created but it won't generalise well on new data)

Level 2: Tidy covariates -> new covariates

- E.g. transformations or other functions of the original covariates
- More necessary for some **methods that may depend a little bit more on the distribution of the data** (e.g. **regression**, **SVMs**) than for others which are less model-based (e.g. classification trees)
- It should be done only on the **training set**. **Building features can only happen in the training set**, not in the test set! You will apply the same transformation then to the covariates in your test set to predict the outcome variable. Otherwise there will be **overfitting problems**!
- The best approach is through **exploratory analysis** (**plotting/tables**)
- New covariates should be added to dataframes

Load Example data

- The original dataset is divded into a training and a test set so that all covariates will be created in the training dataset



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

Common covariates to add, dummy variables

- One idea that is very common when bulding ML algorithms is to **turn covariates that are qualitative** (or **factor variables**) into **dummy** or **indicator variables**.

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

```

 1. Industrial 2. Information
          1090           1012
```

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

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

- It is hard for prediction algorithms to use character variables (such as `jobclass` in the dataset).
- You can deal with this by **transforming the character/qualitative categorical variable into quantitative** using the caret package and the `dummyVars` function.
- For each variable that gets processed by the `dummyVars` function you get N new variables, depending on the number N of categories of the original variable (in this case 2).

Removing zero covariates

- Some of the covariates might have no variability in them, so they are not going to be useful

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

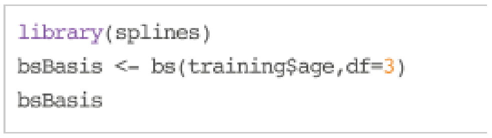
```
           freqRatio percentUnique zeroVar   nzv
year           1.029       0.33302   FALSE FALSE
age            1.122       2.80685   FALSE FALSE
sex            0.000       0.04757    TRUE  TRUE
maritl         3.159       0.23787   FALSE FALSE
race           8.529       0.19029   FALSE FALSE
education      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
```

- You can use the `nearZeroVar` function in caret to identify those variables that have **very little variability** and will likely not be good predictors.

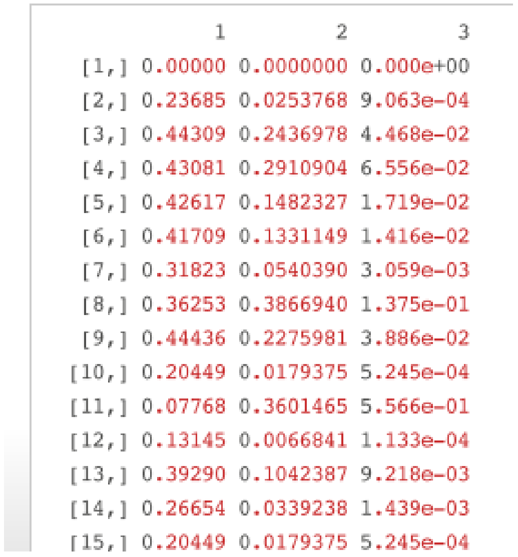
- The metrics of the `nearZeroVar` function tells you the % of unique values for a particular variable (`percentUnique`) and the frequency ratio between the most common value and the second most common value (`freqRatio`)
- You can use these metrics to throw out all variables who have zero or near-zero variance (exclude those with `zeroVar` and `nzv` = `TRUE` from being predictors) since they are less meaningful predictors.

Spline basis

- If you do **linear regression** or **generalised linear regression** as your prediction algorithm, you will be **fitting straight lines** through the data.
- Other times you might want to **fit curvy lines** and you can do this using **basis functions** which you can find in the `splines` package:



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

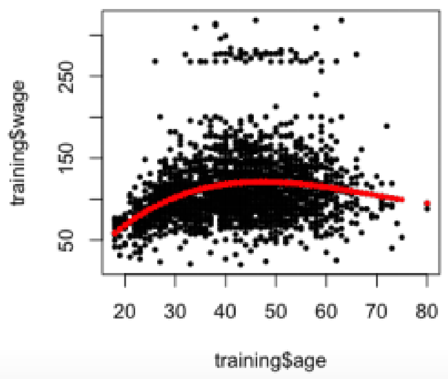


```
             1         2         3
 [1,] 0.00000 0.0000000 0.000e+00
 [2,] 0.23685 0.0253768 9.063e-04
 [3,] 0.44309 0.2436978 4.468e-02
 [4,] 0.43081 0.2910904 6.556e-02
 [5,] 0.42617 0.1482327 1.719e-02
 [6,] 0.41709 0.1331149 1.416e-02
 [7,] 0.31823 0.0540390 3.059e-03
 [8,] 0.36253 0.3866940 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
[15,] 0.20449 0.0179375 5.245e-04
```

- You can use the `bs` function to create a **polynomial variable**.
- `df` specifies the **degree of the polynomial** (in the example, 1st column = age, 2 column = age^2, 3rd column = age^3)

Fitting curves with splines

```
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)
```



- In the example, there is a kind of curvilinear relationship between the variables `age` and `wage`. Using the polynomial terms we created for age, we can fit a curve through the data as opposed to just a straight line.
- In this way you allow more flexibility in the way you model sepcific variables.

Splines on the test set

- On the test set you have to predict the same outcome variables. A critical idea in ML is that **you have to create the covariates in the test dataset using the exact same procedure as the one you used in the training set**.

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

- You are going to use on the test set the same function you used in the training set for creating the polynomial variables (`bsBasis`). In this way you create the polynomial variables for the test set. You cannot apply the `bs` function directly on the predictors (e.g. `age` in test set) of the test set, otherwise you will introduce bias.

Notes and further reading:

- **Level 1 feature creation** (from raw data to covariates):
  - **Science** (or **application-specific knowledge**) is key. E.g try and Google "feature extraction for [data type]". You need to look for the most salient characteristics that are likely to be different between individual samples
  - Err on overcreation of features
  - In some applications (images, voices) automated feature creation is possible/ necessary.
  - **Deep learning** is a way to create features (e.g. for images and voices): https://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! (e.g. features good only on the training set)
- Preprocessing with caret: https://topepo.github.io/caret/pre-processing.html
- If you want to **fit spline models**, use the `gam` method in the caret package which allows **smoothing of multiple variables**, with a different smooth for every variable.
- More on feature creation/data tyding in the Obtaining Data course.

Week 2.7: Preprocessing with Principal Component Analysis

Often you have multiple **quantitative variables** which sometimes might be **highly correlated with each other**. In other words, they will be very similar and almost the excat same variable. In this case, it might not be useful to include each of them in the model, but you might want to include a **summary** that captures most of the information in those quantitative variables.

Correlated predictors

- All the operations of exploration, model creation and feature building will be carried out in the training set only.
- The 58th column is not used for processing because is the outcome:

```
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]))
diag(M) <- 0
which(M > 0.8,arr.ind=T)
```

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

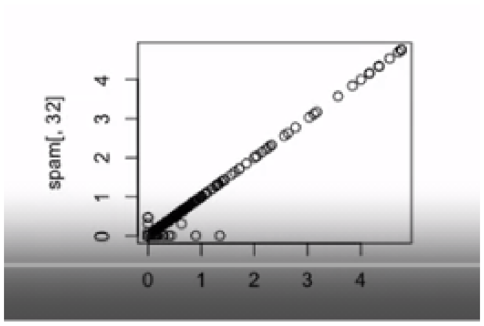
- The correlation between all the covariates is calculated using the `cor` function
- The correlations of covariates with themselves is removed using `diag(M)<-0`

```
names(spam)[c(34,32)]
```

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

```
plot(spam[,34],spam[,32])
```

- The correlation between the 2 variables (`num415` and `num857`) is visible by plotting them one against the other:



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:
    - **Reduce number of predictors**

    - **Reduce noise** (due to averaging of variables that get combined together)

We could rotate the plot

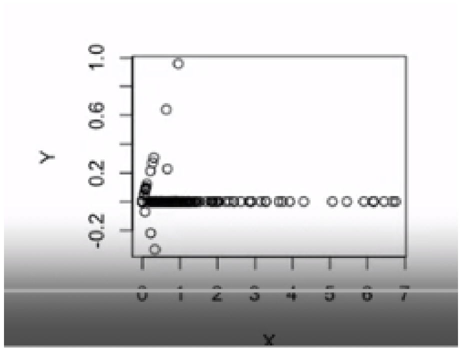
- With PCA you are trying to figure a combination of variables that explains most of their variability.
- As an example, I could either add up the two variables mentioned above (`num415` and `num857`), obtaining a new variable `X`, or I could subtract them, obtaining a new variable `Y`.

$$X = 0.71 \times \text{num415} + 0.71 \times \text{num857}$$

$$Y = 0.71 \times \text{num415} - 0.71 \times \text{num857}$$

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

- If I plot `X` against `Y`:



- As it is visible, most of the variability is happening in the X axis, as there are lots of points spread out across the x-axis. And most of the points are clustered at 0 on the y-axis (i.e. they have a y-value = 0)
- So, adding the two variables captures most of the information contained in them, while subtracting them captures less information --> summing the two variables would be more useful to use as a predictor.

Related problems

- There are two related problems in how you do PCA in a more general sense.
- You have multivariate variables: $X_1, \ldots, X_n = (X_{11}, \ldots, X_{1m})$
    - **Statistical goal**: find a new set of multivariate variables that are uncorrelated and explain as much variance as possible.
    - **Data compression goal**: if you put all the variables together in one matrix, find the best matrix created with fewer variables (lower rank) that explains the original data.
- These two problems are very closely related to each other and they both aim to find fewer variables to explain the most of the original data.

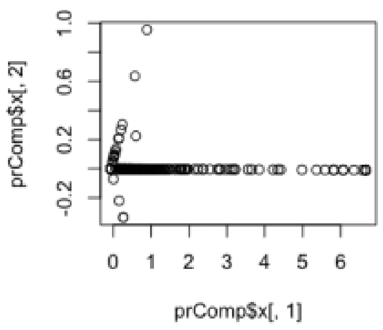
Related solutions: PCA/SVD

- **SVD (Singular Value Decomposition)**: if X is a matrix with each variable in a column and each observation in a row, then SVD is a "matrix decomposition":
$$X = UDV^T$$
Where the columns of U are orthogonal (left singular vectors), **the columns of V are orthogonal (right singular vectors)** and D is a diagonal matrix (singular values)
- **PCA (Principal Component Analysis)**: the **principal components** are equal to the r**ight singular values** if you first **scale** (i.e. subtract the mean and divide by the standard deviation) the variables.

The variables in V are constructed to explain the maximum amount of variation in the data

Principal components in R - `prcomp`

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



- The plot with the 2 principal components is very similar to the one produced above by simply adding or subtracting the two variables.
- Principal components, however, allow you to perform the summary even if you have more than 2 variables that are highly correlated. Using principal components let you look at a large number of quantitative variables and reduce it quite a bit.

- Inside the `prComp` object generated by R you can also have a look at the **rotation matrix** which indicates how the variables are summed up to get the principal components:

```
prComp$rotation
```

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

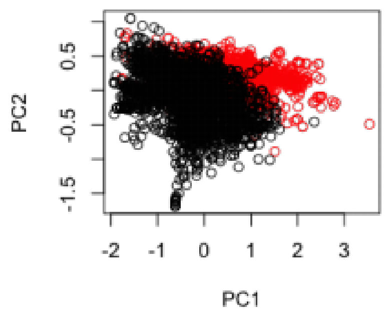
$$PC1 = 0.7081 * num415 + 0.7061 * num857$$

$$PC2 = 0.7061 * num415 - 0.7081 * num857$$

PCA on SPAM data

- PCA can be done on more than 2 variables.
- `typeColor` is going to colour data points in black if they are nonspam or in red if they are spam
- The `prcomp` function calculates principal components on the entire dataset. The transformations to the data applied inside the function (i.e. `log10` and adding `+1`) make the data look a bit more Gaussian (since some fo the variables are skewed). You often need to **normalise** the variables before applying PCA.
- PC1 is no longer a very easy addition between two variables, but it might be a quite complicated combination of all the variables in the dataset. It is the combination that explains most of the variation in the data.
- PC2 is the combination of variables that explains the second most variation, PC3 explains the third most and so forth...

```
typeColor <- ((spam$type=="spam")*1 + 1)
prComp <- prcomp(log10(spam[,-58]+1))
plot(prComp$x[,1],prComp$x[,2],col=typeColor,xlab="PC1",ylab="PC2")
```

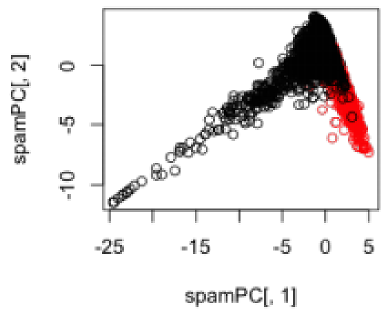


- From the plot you can see that along PC1 there is a little bit of separation of the ham messages (black) from the spam messages (red); the latter ones tend to have a little bit higher values on PC1.
- This is a way to reduce the size of the dataset while still capturing a large amount of variation, which is the main idea behind feature creation.

PCA with caret

- You can PCS in caret as well using the `preProcess` function, using the `method = "pca"` and the number of components to produce (`pcaComp = 2`)
- You then calculate the values of the principal components using caret `predict` function

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



Preprocessing with PCA

- With caret you can fit a model that relates the training variables to the principal components:

```
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)
```

- **In the test dataset you need to use the same principal components you have calculated in the training set**. You do this by passing the pre-process object (`trainPC`) that has been calculated using the training set.

```
testPC <- predict(preProc,log10(testing[,-58]+1))
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
            Specificity : 0.884
```

Alternative (sets # of PCs)

- You can decide not to use the caret `predict` function separately, but build PCA directly into your training exercise.
- Using the caret `training` function you can pre-process the training set using PCA (`preProcess = 'PCA'`)

```
modelFit <- train(training$type ~ .,method="glm",preProcess="pca",data=training)
confusionMatrix(testing$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
            Specificity : 0.915
```

Final thoughts on PCs

- Most useful for **linear-type models** (e.g. linear discriminant analysis, linear and generalised linear regression)
- It can make it **harder to interpret predictors** (since each principal component might be quite a complex weighted sum of the original variables, so could be very hard to interpret)
- Watch out for **outliers**!
  - **Transform** first (with logs/Box Cox)
  - Plot predictors to identify problems

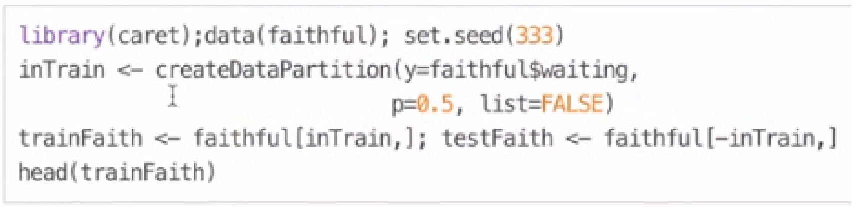
Week 2.8: Predicting with regression

Key ideas

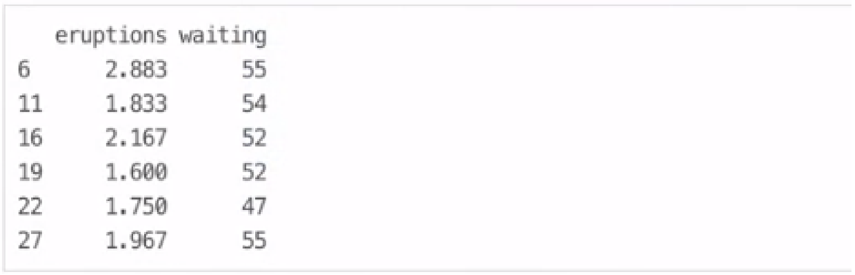
- Fit simple regression (i.e. fitting a line to a set of data by multiplying each predictor by a coefficient and then summing them)
- Plug in new covariates and multiply by the coefficients
- Useful when the linear model is (nearly) correct
- **Pros**:
  - Easy to implement
  - Easy to interpret
- **Cons**:
  - Often poor performance in non-linear settings (it is usually used in combination with other ML algorithms for more complicated scenarios)

Example: old faithful eruptions

We will use a dataset about eruptions of geysers (with data about waiting times between eruptions and time durations of the eruptions themselves).



```r
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)
```

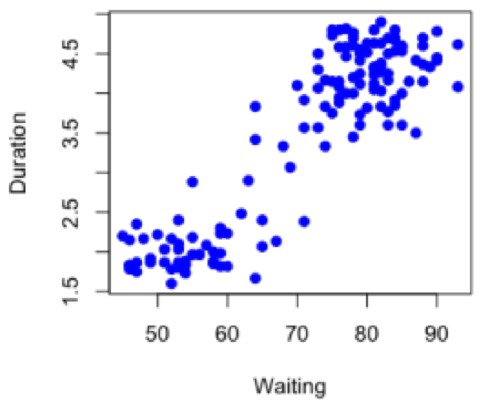


```
   eruptions waiting
6      2.883      55
11     1.833      54
16     2.167      52
19     1.600      52
22     1.750      47
27     1.967      55
```

Eruption duration vs waiting times

- You can see there is roughly a linear relationship between waiting times between eruptions (x-axis) and duration of eruptions (y-axis)

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



Fit a linear model

- ED = Eruption Duration
- WT = Waiting Time

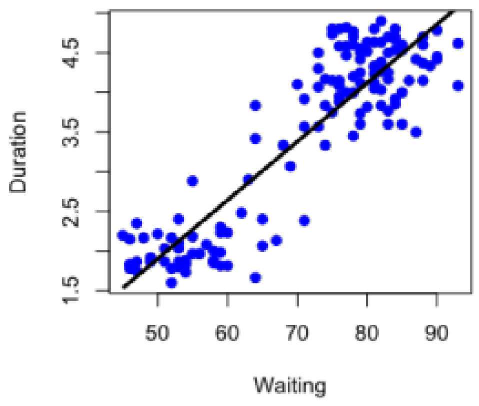
$$ED_i = b_0 + b_1 WT_i + e_i$$

- The points never exactly follow a line, this is why we allow for some error (e_i) in the model

Model fit

- You can plot the regression line that fits the model data using the fitted values generated by the model (available as `lm1$fitted`).

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



Predict a new value

- This can be done using the model estimated parameters for b_0 and b_1
- We cannot add the error term because is unknown

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

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

```
(Intercept)
      4.119
```

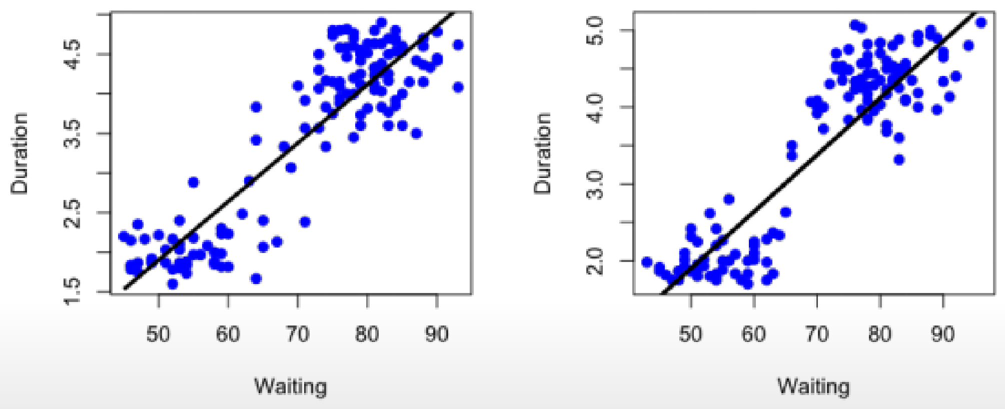
```
newdata <- data.frame(waiting=80)
predict(lm1,newdata)
```

```
    1
4.119
```

Plot predictions: training and test set

- On the test set, the regression line does not fit the data as well as on the training set (this is to be expected, since the model was trained on the training data).

```
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 and test set errors

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

```
[1] 5.752
```

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

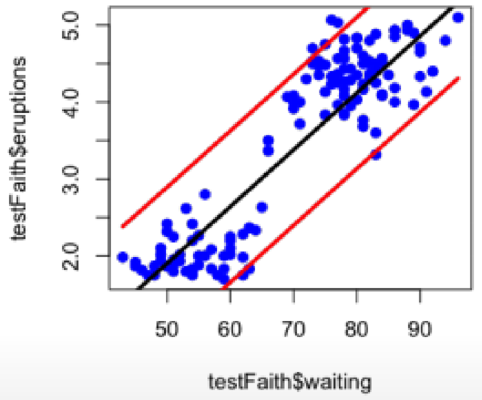
```
[1] 5.839
```

- The RMSE on the test set is more realistic (and usually always bigger) than the one obtained on the training set. This is because the test error contains also the added error variability when the model is used on a new dataset that has not seen during training.

Prediction intervals

- When using **linear modelling** for prediction, you can calculate prediction intervals.
- The linear model in R can provide prediction intervals by specifying the parameter `interval = "prediction"`

```
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)
```



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
Multiple R-squared:  0.803, Adjusted R-squared:  0.802 
```

Week 2.9: Predicting with regression - Multiple covariates

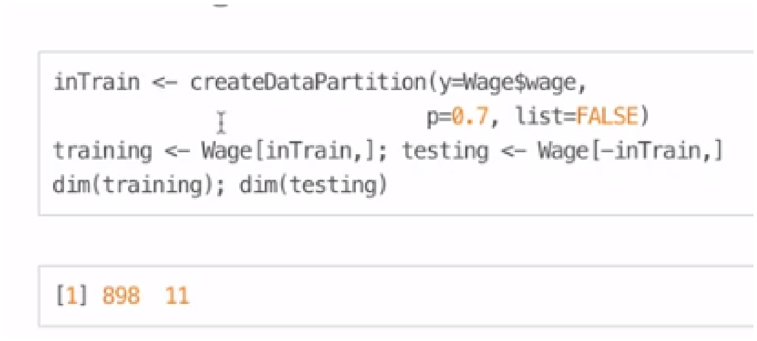
Example: Wage data

```
library(ISLR); library(ggplot2); library(caret);
data(Wage); Wage <- subset(Wage,select=-c(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                                              
                           (Other)              :   0                                              
```

- We exclude `logwage` as it is the variable we are trying to predict

- To do a bit more of exploration, we need to divide the dataset into training and test set



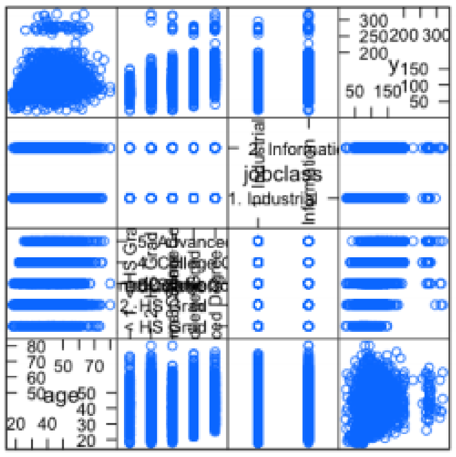
```
inTrain <- createDataPartition(y=Wage$wage,
                               p=0.7, list=FALSE)
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
dim(training); dim(testing)
```

```
[1] 898  11
```

Feature plot

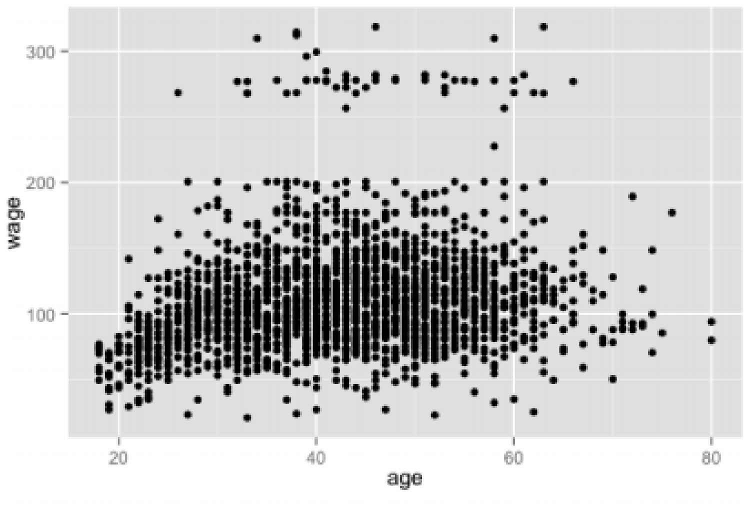
- It shows how variables are related to each other
- For the `jobclass` covariate, the `information` group seems to have a slightly better income
- For the `age` covariate, there are some outliers which define two groups

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

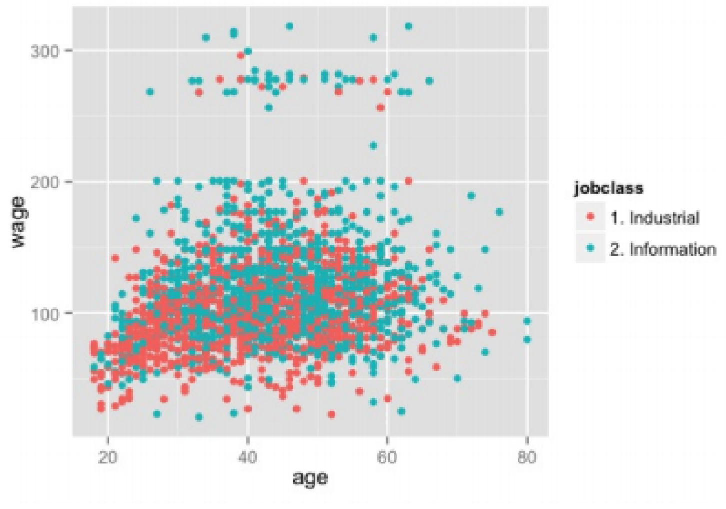


Scatter Plot Matrix

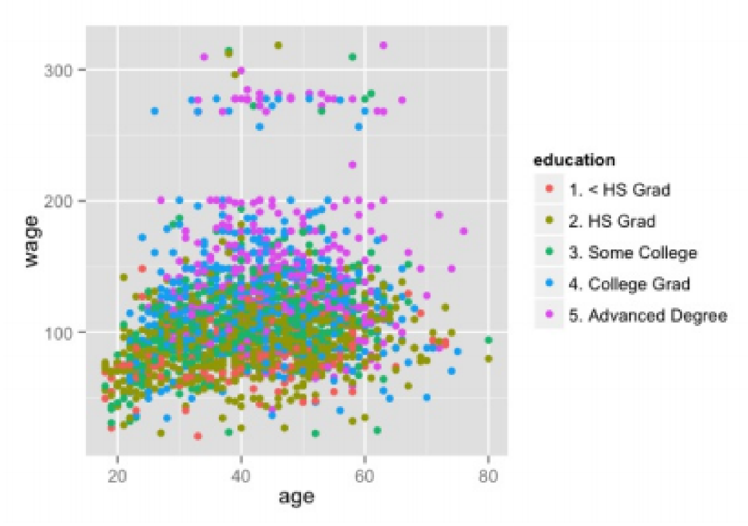
```
qplot(age,wage,data=training)
```



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



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



Fit a multiple linear regression model

- Cateogrical covariates can be expressed in the model as **indicators** (each level of an indicator is given either a 0 or a 1)

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

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

```
Linear Regression

2102 samples
  11 predictors

No pre-processing
Resampling: Bootstrapped (25 reps)

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

Resampling results
```

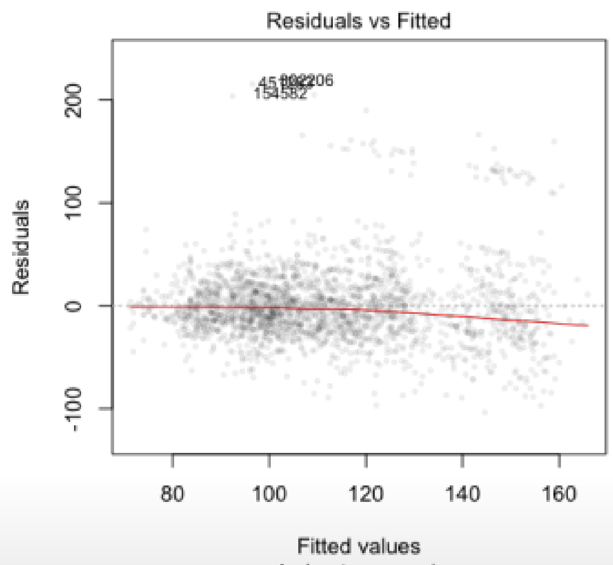
- We fit the model using the caret package

- `jobclass` and `education` are factor variables, so by default caret creates the indicator variables as predictors.

Diagnostics

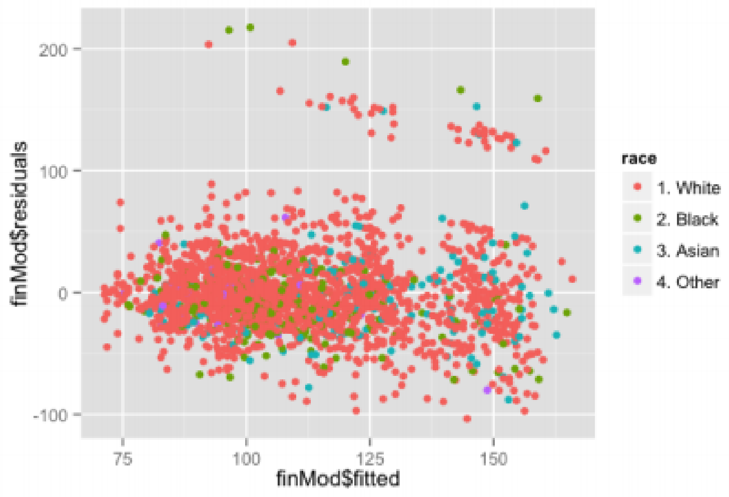
- We can look at some diagnostic plots
- You can predict the model fitted values versus the residuals.
- The ideal would be to have the points equally spread along y = 0

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



- You can also use the plot above and see if any other covariate not included in the model could explain the residuals:

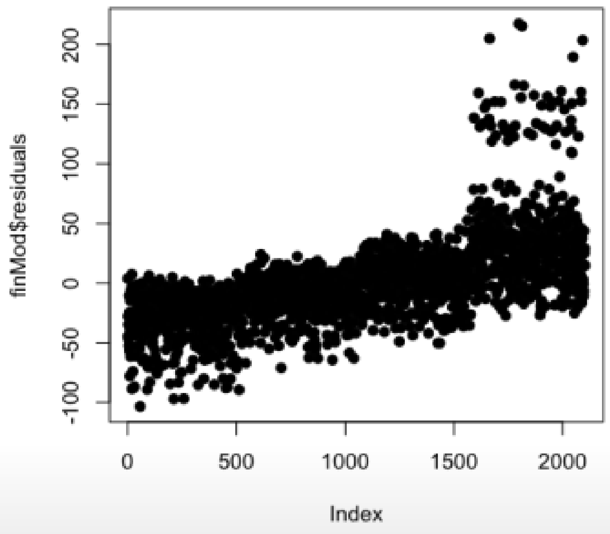
```
qplot(finMod$fitted,finMod$residuals,colour=race,data=training)
```



Plot residuals by index

- The highest residuals seem to be associated to the rows with the highest number

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

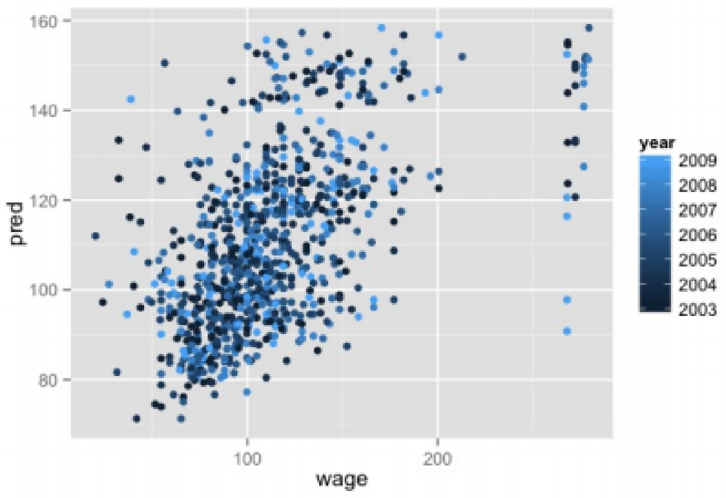


- This suggests that there is a variable missing from the model, since the residuals should not have any relationship with the row order.
- There might be a relationship with a time variable according to which the rows of the dataset are ordered by.

Predicted vs truth in test set

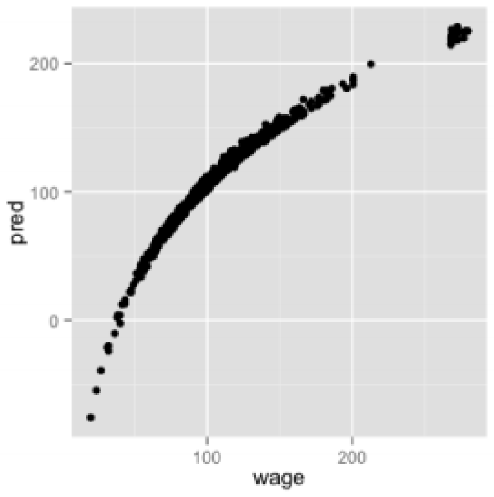
- Ideally the predictions in the test set would be very close to the target.
- This plot might help you to see if there is any trend you might have missed in the model
- However, when exploring the test set, you cannot go back and retrain the model. It is more a check to evaluate whether your model analysis has worked or not.

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



Using all covariates in the model building

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


pred
wage