Week 2: The `caret` package

Lectures*: 1. The `caret` package, 2. Data slicing, 3. Training options, 4. Plotting predictions, 5. Basic preprocessing, 6. Covariate creation, 7. Preprocessing with PCA, 8. Predicting with regression, 9. Predicting with Regression Multiple Covariates

Juan David Leongómez

17 March, 2021

Contents

*All lectures by Jeffrey Leek (John Hopkins Bloomberg Scool of Public Health).

---

1 Lecture 1: `caret` package

The `caret` package (Kuhn, 2020b, currently v6.0-86) 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 (manual: Kuhn, 2019, R documentation: 2020a).

1.1 `caret` functionality

- Some preprocessing (cleaning)
  - `preProcess`
- Data splitting (for cross-validation)
  - `createDataPartition`
  - `createResample`
  - `createTimeSlices`
- Training/testing functions
  - `train`
  - `predict`
- Model comparison (to see how well the models did in new datasets)
  - `confusionMatrix`

1.2 Machine learning algorithms in `R`

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
- Regression
- Naive Bayes
- Support vector machines
- Classification and regression trees
- Random forests
- Boosting
- etc.

All of these algorithms are built by a variety of different developers, all coming from different backgrounds, so the interfaces that each of these sort of prediction algorithms is slightly different. Because of this, objects from different algorithms have different classes (see Fig. 1). In order to `predict`, is therefore usually necessary to add the `type` parameter.

```
knitr::include_graphics("caret_objects.png")
```

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)

Figure 1: Different caret objects.

1.3 SPAM example: Data splitting

This a simple example. Here, after loading the packages and data, we partition the data into training and test sets. Here, we used 75% of our data to train the model and 25% to test, using the `createDataPartition` function.

1.3.1 Sub-setting into training (75%) and testing (25%) datasets (`createDataPartition`)

Then, we create subsets for training and testing based on that partition

```
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
library(kernlab)
```

```
##
## Attaching package: 'kernlab'
```

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

```
data(spam)

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

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

1.3.2 Fit model (`train`)

Now, we can fit a model using the `train` function. We used all variables (`~.`) as predictors of `type` (spam, nonspam). In this case, it used bootstrapping with 25 replicates, correcting for the potential bias that might come from bootstrap sampling.

```
RNGkind(sample.kind = "Rounding") #to make seed equivalent to older R versions
set.seed(32343)

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

```
## Generalized Linear Model
##
## 3451 samples
##   57 predictor
##    2 classes: 'nonspam', 'spam'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 3451, 3451, 3451, 3451, 3451, 3451, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9193943  0.8301599
```

To look at the final model, you can look at the `$finalModel` component of the model fit.

```
modelFit$finalModel
```

```
##
## Call:  NULL
##
## Coefficients:
##       (Intercept)               make            address                all
##         -1.575027          -0.425783          -0.178709          -0.044971
##             num3d                our               over             remove
##          2.044847           0.539793           0.637509           1.970038
##          internet              order               mail            receive
##          0.839051           0.998456           0.284555          -0.328226
##              will             people             report          addresses
##         -0.112554           0.015602           0.132370           0.742217
##              free             business              email                you
##          1.081126           0.883238           0.380485           0.112489
##            credit                your               font             num000
##          0.918036           0.266757           0.210970           3.140052
##             money                   hp                hpl             george
##          0.660528          -2.380621          -0.592496         -12.930757
##            num650                  lab               labs             telnet
##          0.355340          -2.130911          -0.241039          -3.678964
##            num857                data             num415              num85
##          4.135502          -1.146126           0.850061          -1.626312
##         technology             num1999              parts                 pm
##          1.097369           0.015770           1.125749          -0.786740
##            direct                   cs            meeting           original
##         -0.358573         -53.145453          -2.941755          -1.053241
##           project                   re                edu              table
##         -1.689666          -0.820989          -1.811544          -4.251289
##         conference      charSemicolon   charRoundbracket  charSquarebracket
```

```
##         -4.363114          -1.265326          -0.415305          -0.841217
##   charExclamation         charDollar           charHash         capitalAve
##          0.217346           4.354545           1.860485           0.042716
##       capitalLong       capitalTotal
##          0.004654           0.001255
##
## Degrees of Freedom: 3450 Total (i.e. Null);  3393 Residual
## Null Deviance:       4628
## Residual Deviance: 1339  AIC: 1455
```

1.3.3 Predict with the new model (predict)

Then, you can predict on a new samples by using the `predict` function (here, only the first 100 predictions).

```
predictions <- predict(modelFit,
                       newdata = testing)
predictions[1:100]
```

```
##   [1] spam    spam    spam    spam    nonspam spam    spam    spam    spam
##  [10] nonspam spam    spam    nonspam nonspam spam    spam    spam    spam
##  [19] spam    spam    spam    spam    spam    spam    spam    spam    spam
##  [28] spam    spam    spam    spam    spam    spam    spam    spam    spam
##  [37] spam    spam    spam    spam    spam    spam    spam    nonspam spam
##  [46] spam    spam    spam    spam    spam    spam    nonspam spam    nonspam
##  [55] spam    spam    spam    nonspam spam    spam    spam    spam    spam
##  [64] spam    spam    spam    spam    spam    spam    spam    spam    spam
##  [73] spam    nonspam spam    spam    spam    spam    spam    spam    spam
##  [82] nonspam nonspam spam    spam    spam    spam    spam    spam    nonspam
##  [91] spam    spam    spam    spam    nonspam spam    spam    spam    spam
## [100] spam
## Levels: nonspam spam
```

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. One way that you can do that is by calculating the confusion matrix (function `confusionMatrix`), so that's using this confusion matrix function, and so note the capital M here.

1.3.4 Confusion matrix (evaluate with confusionMatrix)

To evaluate the prediction from our model, we can compare the predictions on the testing set, against the actual outcome variable in that testing set.

```
confmat <- confusionMatrix(predictions, testing$type)
confmat
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction nonspam spam
##    nonspam     659   46
##    spam         38  407
##
##                Accuracy : 0.927
##                  95% CI : (0.9104, 0.9413)
##     No Information Rate : 0.6061
##     P-Value [Acc > NIR] : <2e-16
##
```

```
##                  Kappa : 0.8466
##
##  Mcnemar's Test P-Value : 0.445
##
##             Sensitivity : 0.9455
##             Specificity : 0.8985
##          Pos Pred Value : 0.9348
##          Neg Pred Value : 0.9146
##              Prevalence : 0.6061
##          Detection Rate : 0.5730
##    Detection Prevalence : 0.6130
##       Balanced Accuracy : 0.9220
##
##        'Positive' Class : nonspam
##
```

This creates a table for which of the cases that you predicted to be nonspam are actually nonspam, which is the number of cases where it was spam, and you predicted to be spam and so forth.

```
kable(confmat$table,
      booktabs = TRUE) %>%
  kable_styling(position = "center", latex_options = "HOLD_position")
```

	nonspam	spam
nonspam	659	46
spam	38	407

And then it gives you a bunch of summary statistics like Sensitivity, Specificity, PPV, NPV, Accuracy:

```
kable(confmat$byClass,
      booktabs = TRUE,
      col.names = "Value") %>%
  kable_styling(position = "center", latex_options = "HOLD_position")
```

	Value
Sensitivity	0.9454806
Specificity	0.8984547
Pos Pred Value	0.9347518
Neg Pred Value	0.9146067
Precision	0.9347518
Recall	0.9454806
F1	0.9400856
Prevalence	0.6060870
Detection Rate	0.5730435
Detection Prevalence	0.6130435
Balanced Accuracy	0.9219677

1.3.4.1 Variable importance (*not in the lectures so far*)

Variable importance can be obtained using `varImp` on the model fitted using `train`. The `data.frame` it contains can be plotted in `ggplot`.

```
library(ggpubr)

imp.df <- varImp(modelFit)$importance
ggplot(imp.df, aes(x = reorder(row.names.data.frame(imp.df), Overall),
                   y = Overall,
                   fill = Overall)) +
  geom_bar(stat="identity", position="dodge") +
  coord_flip() +
  ylab("Variable Importance") +
  xlab("") +
  ggtitle("Information Value Summary") +
  scale_fill_gradient(low="blue", high="red") +
  theme_pubclean()
```

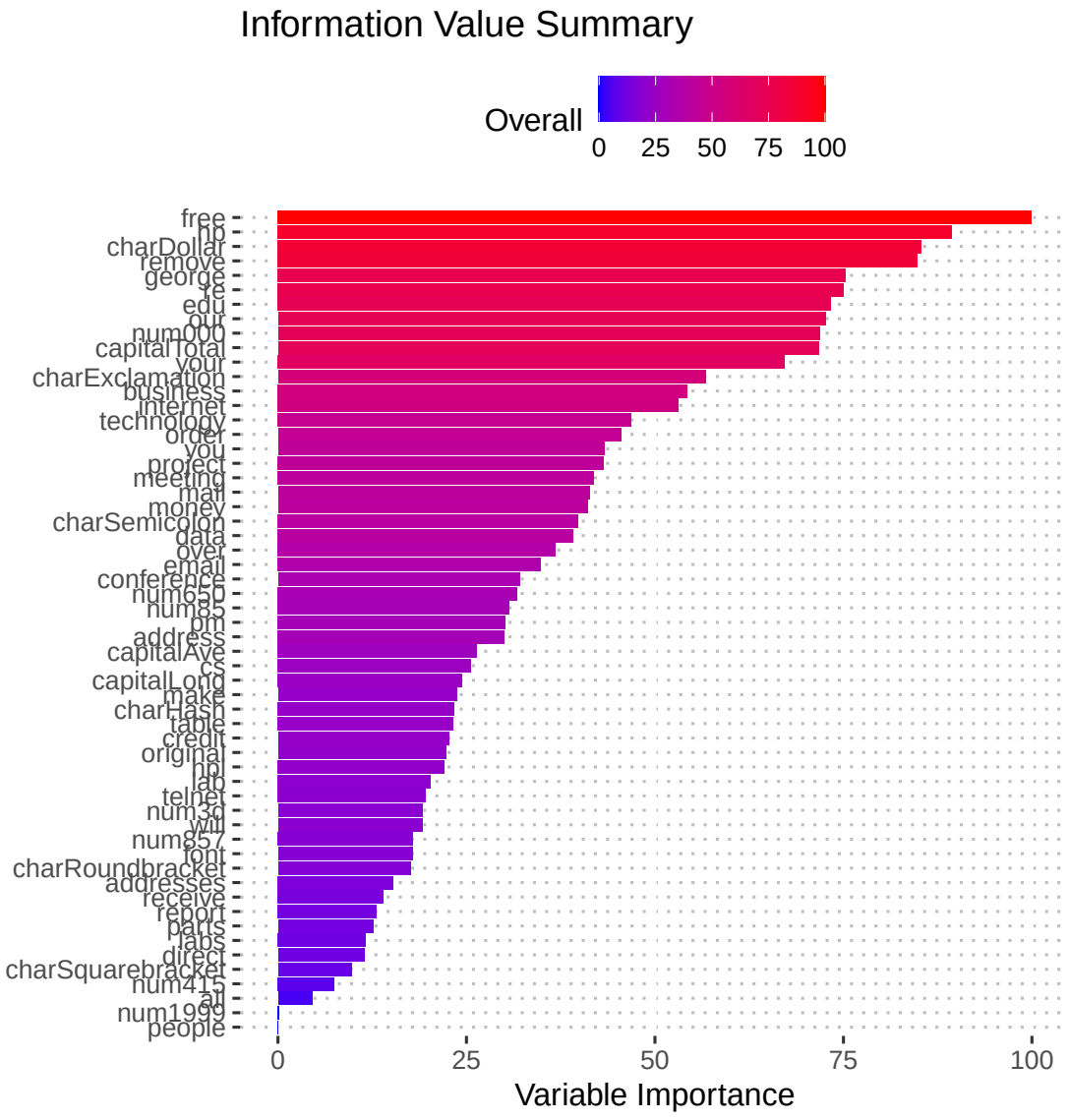

Figure 2: Importance of the variables from the spam model.

1.4 Recommended for further information

- (Kuhn, n.d.)
- (Kuhn, 2013)
- (Kuhn, 2008)

---

2 Lecture 2: Data slicing

This lecture is about data slicing; you may use data slicing either for:

1. Building your training and testing sets right at the beginning of your prediction function creation, or
2. Performing cross validation or boot strapping within your training set, in order to evaluate your models

2.1 SPAM example: Data splitting

Again, we use the `spam` dataset from the `kernlab` package, and split it to 75% for training, and 25% for testing.

```
library(caret)
library(kernlab)
data(spam)


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

2.2 SPAM example: k-fold

We can also split the data for k-fold cross-validation, and just include the k number, using the `createFolds` function.

```
RNGkind(sample.kind = "Rounding") #to make seed equivalent to older R versions
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
```

This splits the samples in order. For example, if we look at the first 10 elements of the fist sample:

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

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

Alternatively, instead of returning the train sets (`returnTrain = TRUE`), you can ask the function to return the test sets (`returnTrain = FALSE`).

```
RNGkind(sample.kind = "Rounding") #to make seed equivalent to older R versions
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
```

Again, this splits the samples in order. For example, if we look at the first 10 elements of the fist sample:

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

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

2.3 SPAM example: Resampling

Resampling can be done using the `createResample` function, and specifying how many times you want to resample the data (e.g. `times = 10`).

```
RNGkind(sample.kind = "Rounding") #to make seed equivalent to older R versions
set.seed(32323)

folds <- createResample(y = spam$type,
                        times = 10,
                        list = TRUE)

sapply(folds, length)
```

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

And look at the elements selected. In this case, because we are resampling with replacement, the same element can be repeated:

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

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

2.4 SPAM example: Time Slices

Time slices can be done using the `createTimeSlices` function, and specifying that, for example, I want to create slices that have a window of about 20 samples in them (`initialWindow = 20`), and I want to say I'm going to predict the next 10 samples (`horizon = 10`)out after I take the initial window of 20.

```
RNGkind(sample.kind = "Rounding") #to make seed equivalent to older R versions
set.seed(32323)

tme <- 1:1000
folds <- createTimeSlices (y = tme,
```

```
                           initialWindow = 20,
                           horizon = 10)

names(folds)
```

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

If we look at the elements selected for training (`initialWindow`), we can see the first 20 elements have been selected for training (`folds$train`):

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

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

And that the next 10 elements were selected for testing (`folds$test`):

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

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

More information on Time Slicing is found on the recommended lectures (Kuhn, n.d., 2013, 2008).

---

3 Lecture 3: Training options

This is a brief lecture about some of the training control options that you have when training while using the caret package. For this lecture, we're going to be using the spam example again just to illustrate how these ideas work.

```
library(caret)
library(kernlab)
data(spam)

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

Usually, what you would do when you fit a model is basically just use the train function, where you basically set all of the defaults to be whatever defaults that the train function chooses for you, specifying only the method (`method = "glm"`) that you're going to be using to fit and which data set you're going to be using (`data = training`).

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

3.1 Train options

You can also use a large set of option for training.

```
args(train)

function(x, y,
         method = "rf",
         preProcess = NULL, ...,
```

```
          weights = NULL,
          metric = ifelse(is.factor(y), "Accruracy", "RMSE"),
          maximise = ifesle(metric = "RMSE", FALSE, TRUE),
          trControl = trainConrol(),
          tuneGrid = NULL,
          tuneLength = 3)
```

3.1.1 Metric options (`metric`)

Continuous outcomes:

- `RSME` = Root Mean Squared Error
- `RSquared` = R^2 from regression models

Categorical outcomes

- `Accuracy` = Fraction correct
- `Kappa` = A measure of concordance

3.1.2 Train control (`trainControl`)

The `trainControl` argument allows you to be much more precise about the way that you train models.

```
args(trainControl)

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

```
                          alpha = 0.05,
                          method = "gls",
                          complete = TRUE),
          trim = FALSE, allowParallel = TRUE)
```

3.1.3 Train control (`trainControl`) resampling

- `method`
  - `"boot"` = bootstrapping
  - `"boot632"` = bootstrapping with adjustemnt
  - `"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 considerably slow things down

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.

3.1.4 Setting the seed

- It is often useful to set an overall seed
- For parallel fits, you can also set a seed for each sample (and it is most useful in this case)

```
RNGkind(sample.kind = "Rounding") #to make seed equivalent to older R versions
set.seed(1235)

modelFit2 <- train(type ~.,
                   data = training,
                   method = "glm")
modelFit2
```

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

```
##   0.9091913  0.8095559
```

3.2 Further resources

- Caret tutorial (2008)
- Model training and tuning (2019)

---

4 Lecture 4: Plotting predictions

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 plotting the data, and in particular plotting the predictors.

We will use the `Wage` dataset, from the `ISLR` package, from the book *Introduction to Statistical Learning* (James et al., 2013).

```
library(ISLR)
library(ggplot2)
library(ggpubr)
library(caret)

data(Wage)
summary(Wage)
```

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

Just by looking at the `summary` of the data, some interesting characteristics are obvious; for example, that the sample is only of males.

Get training/test sets

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

training <- Wage[inTrain,]
testing <- Wage[-inTrain,]
dim(training); dim(testing)
```

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

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

4.1 Feature plot (caret package)

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

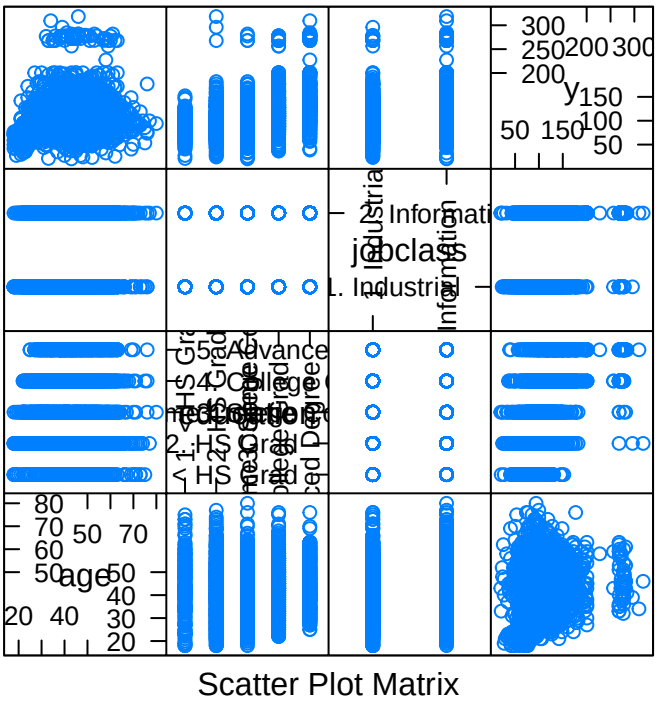

Figure 3: The `featurePlot` from the `caret` package. In this case, we asked for the association of three predictors (`age, education, jobclass`) with the dependent variable (`wage`), as well as amongst them. In particular, you need to look at the first row (i.e. the assiociations between the predictors and the outcome variable); in this case, for example, there is a clear association between `ajobclass` and `wage`.

4.2 Qplot (*ggplot2* package)

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

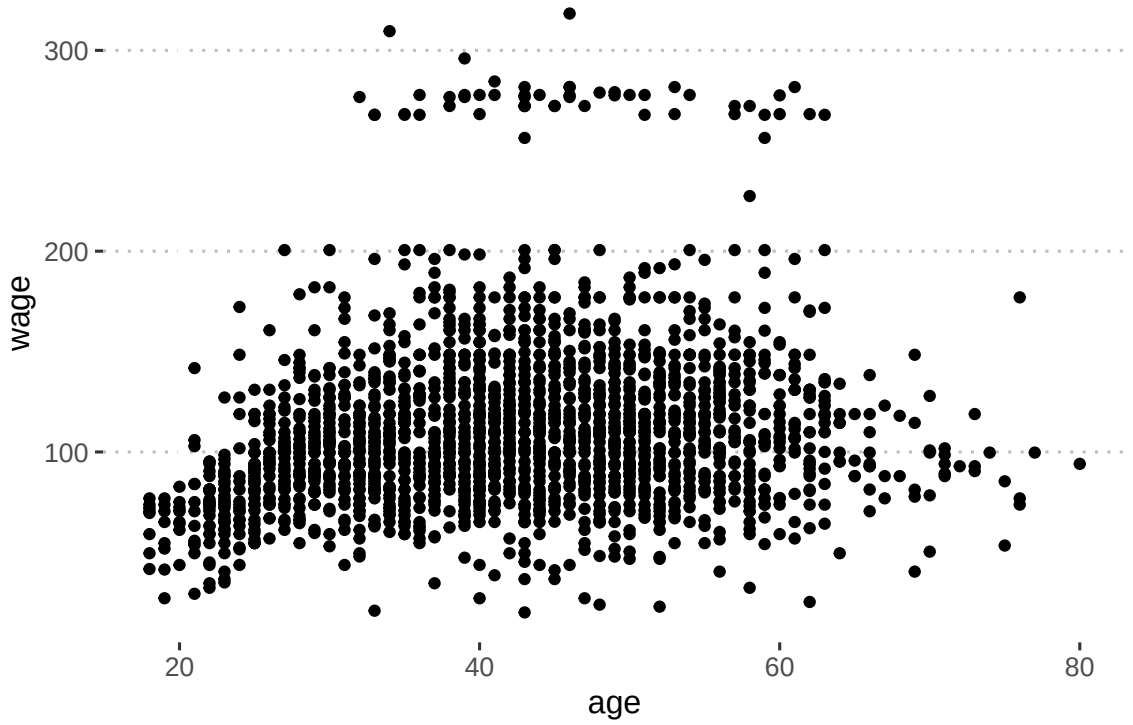

Figure 4: The `qplot` from the `ggplot2` package. In this case, we asked for the association between `age` and `wage`.

While there seems to be a clear association, it is also apparent that there are two `chunks`; the bottom one (with most of the data) shows some association, but the top one is clearly different. To try to understand this weird situation, we can plot by another variable (in this case, `jobclass`).

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

Figure 5: The `qplot` from the `ggplot2` package. In this case, we asked for the association between `age` and `wage` by `jobclass`.

This shows that most of the points in the *weird* chunk, are for people on *information* jobs. This gives you a way to detect variables that might be important in your model, as they show, variation in the data.

4.3 Add regression smoothers (*ggplot2* package)

```
qplot(age, wage, colour = education, data = training) +
  geom_smooth(method = "lm") +
  theme_pubclean()
```

```
## `geom_smooth()` using formula 'y ~ x'
```

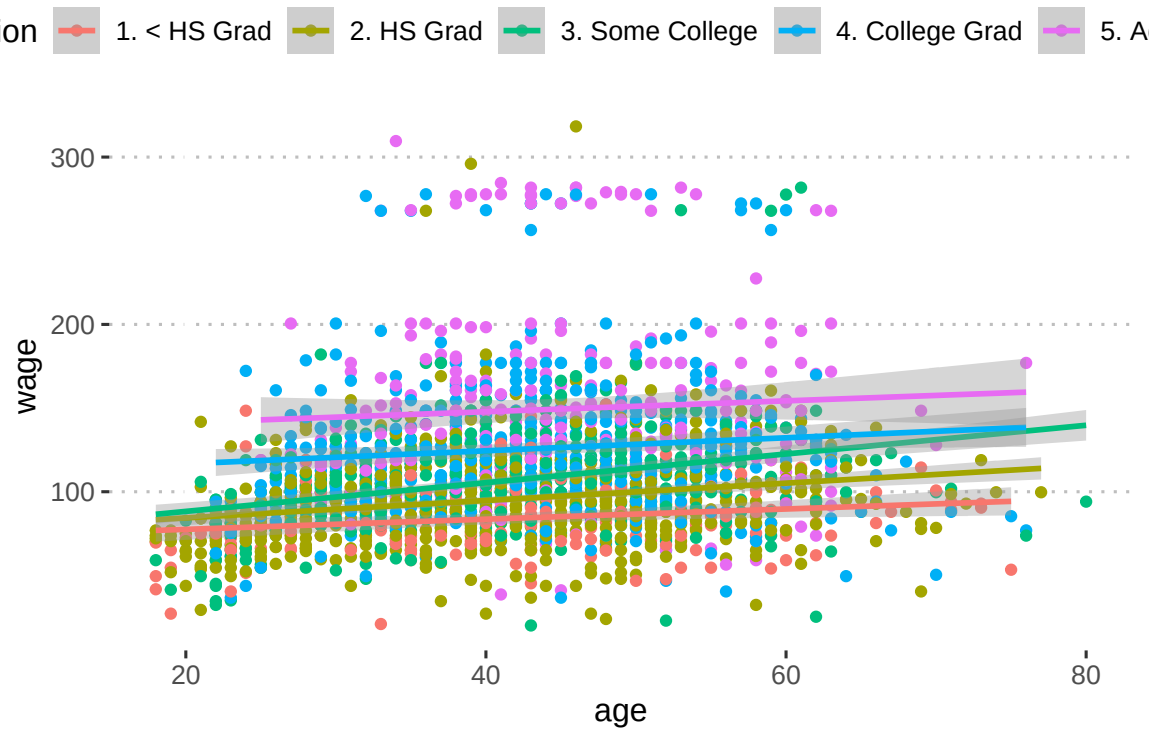

Figure 6: The `qplot` from the `ggplot2` package. In this case, we asked for the association between `age` and `wage` by `education`, including regression lines for each `education` level.

4.4 cut2, making factors (Hmisc package)

the `cut2` function from the `Hmisc` package, allows us to break up things like the wage variable into different categories cause sometimes it's clear that specific categories seem to have different relationships.

```
library(Hmisc)
```

```
## Loading required package: survival
```

```
##
## Attaching package: 'survival'
```

```
## The following object is masked from 'package:caret':
##
##     cluster
```

```
## Loading required package: Formula
```

```
##
## Attaching package: 'Hmisc'
```

```
## The following objects are masked from 'package:base':
##
##     format.pval, units
```

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

```
## cutWage
## [ 20.1, 91.7) [ 91.7,118.9) [118.9,318.3]
##           702           731           669
```

Dividing a predictor into factor levels, can help understand trends. For example:

```
qplot(cutWage, age, fill = cutWage, data = training, geom = "boxplot") +
  theme_pubclean()
```

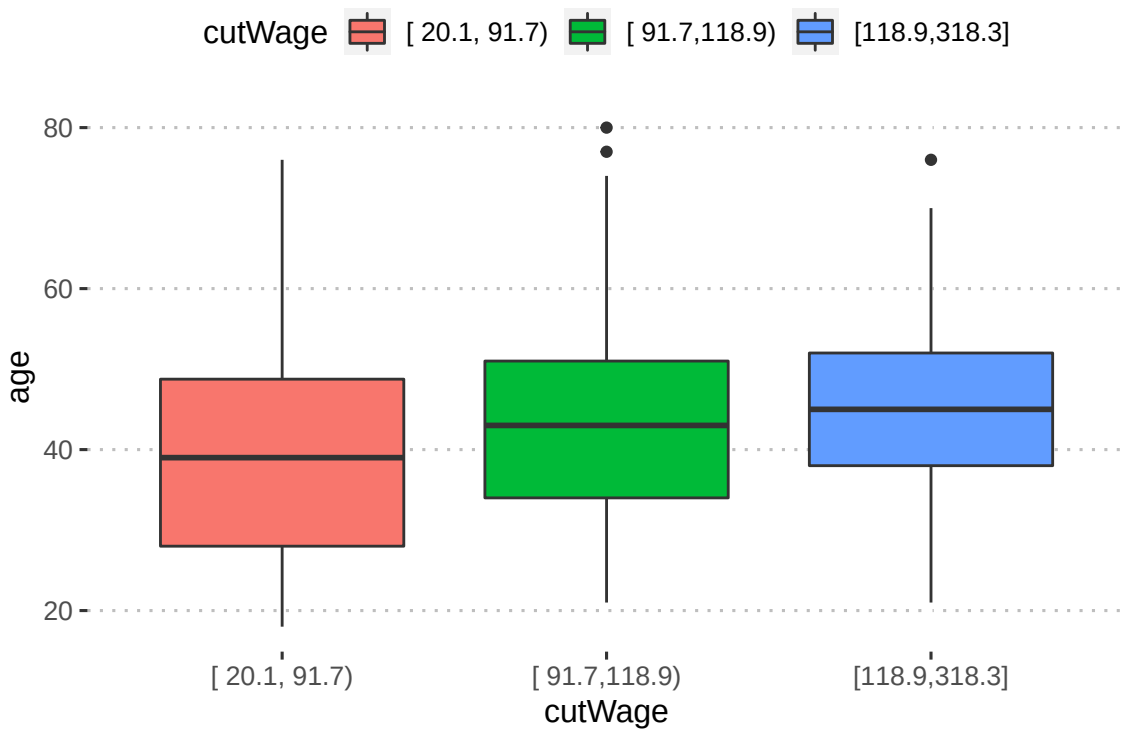

Figure 7: Boxplots `qplot` for the association between `age` and our new 3-level factor `cutWage`.

Adding the point can also help how many there are in each category. In the class they use `jitter` without adding an `alpha` value to these, so they make two panels. Here, I used `alpha` to the jittered points, so panel B should be enough.

```
library(ggpubr)

p1 <- qplot(cutWage, age, fill = cutWage, data = training, geom = "boxplot") +
  theme_pubclean()
p2 <- p1 +
  geom_jitter(alpha = .2) +
  theme_pubclean()

ggarrange(p1, p2,
          ncol = 2,
          common.legend = TRUE,
          legend = "bottom",
          labels = "AUTO")
```

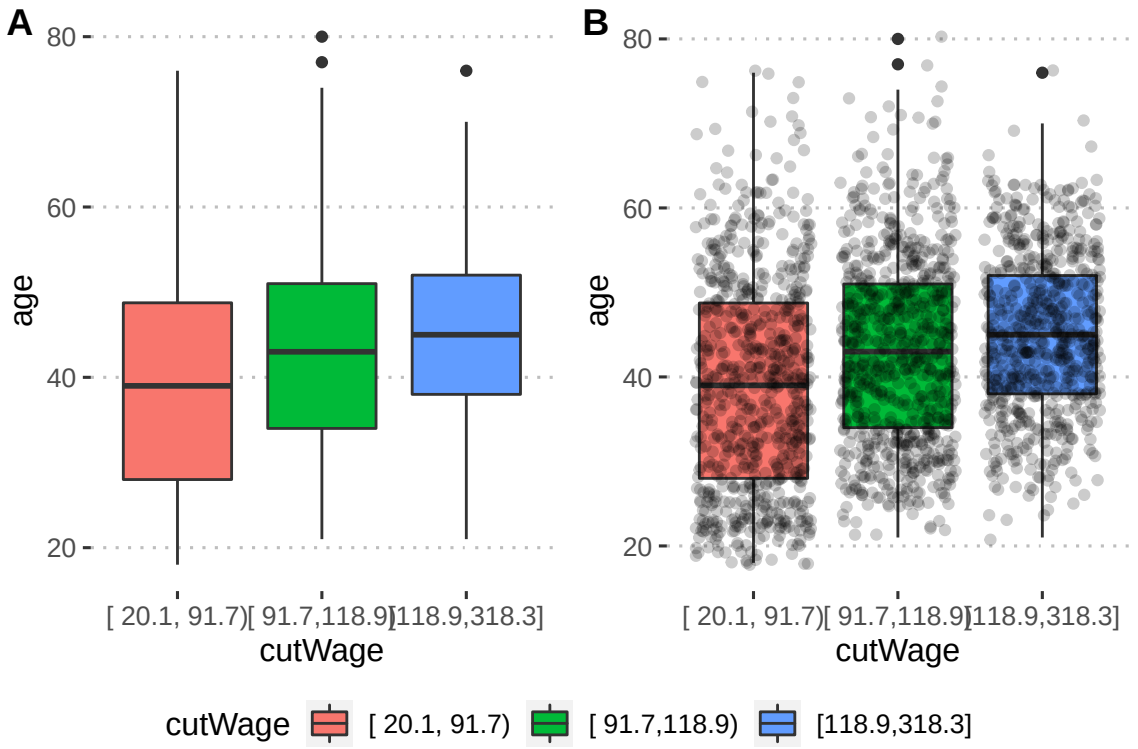

Figure 8: Boxplots `qplot` for the association between `age` and our new 3-level factor `cutWage`. **A.** Boxplots without points. **B.** Boxplots with jittered and somewhat transparent points.

Because there are lots of points in each `cutWage` category, any trend we see is likely real.

4.5 Tables

Another useful option is to look at tables. For example:

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

```
##
## cutWage         1. Industrial 2. Information
##   [ 20.1, 91.7)           442            260
##   [ 91.7,118.9)           378            353
##   [118.9,318.3]           258            411
```

It is clear that, for example, there are more industrial jobs in the lower wage variable than there are information jobs. And that trend reverses itself for the highway jobs. This can also be converted to proportions using the `prop.table` function (the number **1**, refers to proportions per row; a number 2 would calculate proportions per column):

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

```
##
## cutWage         1. Industrial 2. Information
##   [ 20.1, 91.7)     0.6296296      0.3703704
##   [ 91.7,118.9)     0.5170999      0.4829001
##   [118.9,318.3]     0.3856502      0.6143498
```

4.6 Density plots

Another way to look at the data is with densities. Interestingly, in this case there is a second peak in high wages for people with more education.

```
qplot(wage, color = education, data = training, geom = "density") +
  theme_pubclean()
```

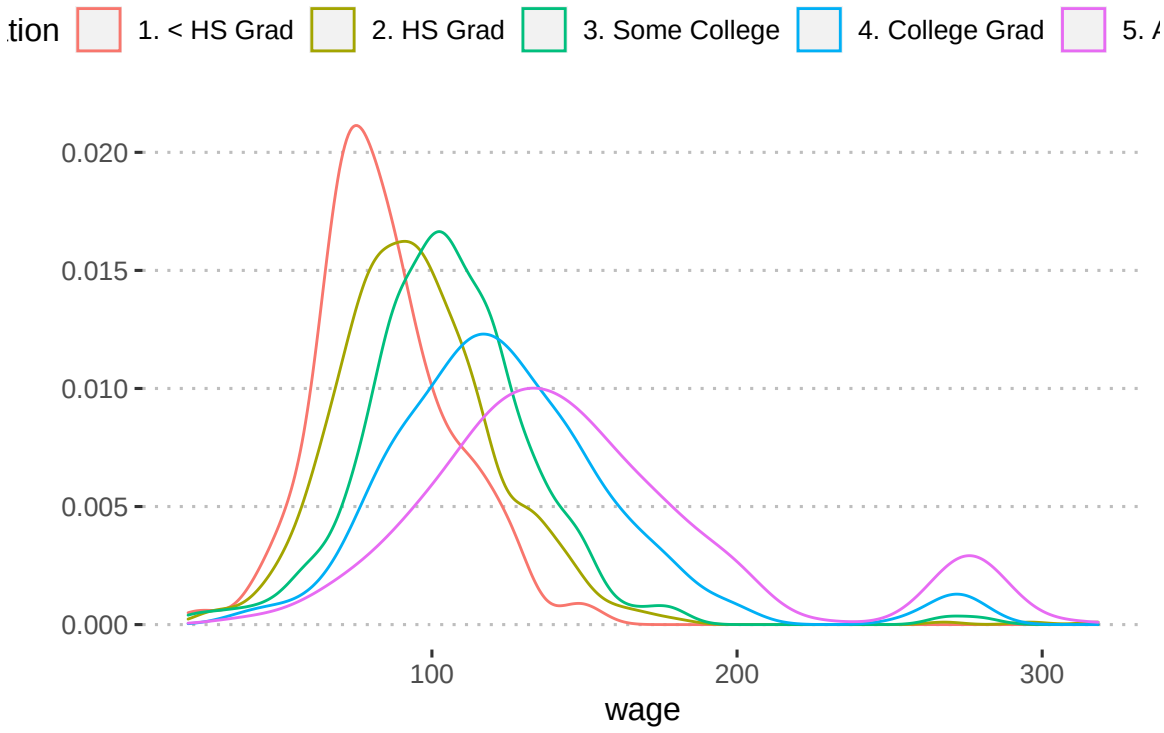

Figure 9: Density `qplot` according to education level.

But (and this is MY idea), it is mainly for people in information jobs: interestingly, there is a second peak for people with more education.

```
qplot(wage, color = education, data = training, geom = "density") +
  facet_wrap(~jobclass) +
  theme_pubclean()
```

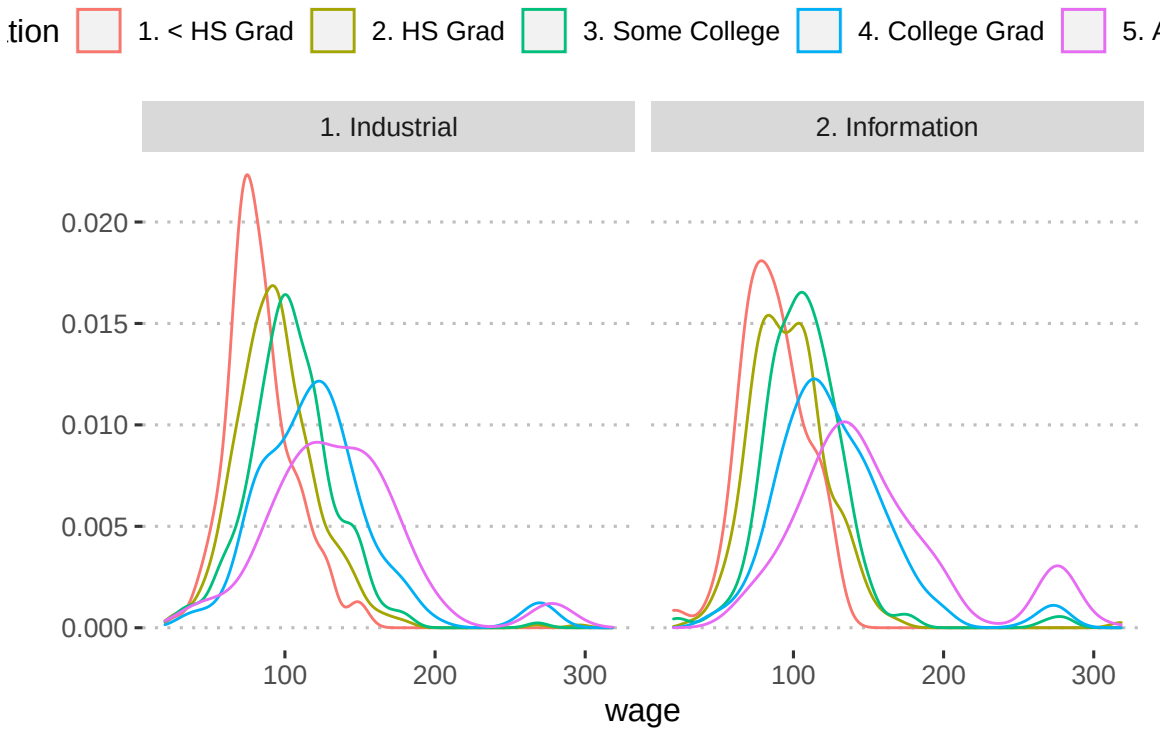

Figure 10: Density `qplot` according to education level, split by `jobclass`.

4.7 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**
  1. Imbalance in outcomes/predictors
  2. Outliers
  3. Groups of points not explained by a predictor
  4. Skewed variables

Further Reading

- ggplot2 tutorial
- caret visualizations

---

5 Lecture 5: Basic preprocessing

This lecture's about preprocessing predictor variables. It is important to plot the variables upfront so you can see if there's any sort of weird behaviour of those variables. Sometimes predictors will look very strange or the distribution will be very strange, and you might need to transform them in order to make them more useful for prediction algorithms.

This is particularly true when you're using model based algorithms, such as linear discriminate analysis, naive Bayes, or linear regression. Pre-processing can be more useful often when you're using model based approaches, then when you're using more non parametric approaches.

5.1 Why pre-process?

When deciding how to preprocess data, or how to explore data, we **only look at the training set**.

```
library(caret)
library(kernlab)
library(ggplot2)
library(ggpubr)

data(spam)

inTrain <- createDataPartition(y = spam$type,
                               p = 0.75,
                               list = FALSE)

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

gghistogram(training, x = "capitalAve",
            fill = "blue",
            color = "blue",
            add = "mean",
            rug = TRUE,
            bins = 12,
            palette = "blue",
            add_density = TRUE) +
  labs(x = "Average capital run length",
       y = "Frequency") +
  theme_pubclean()
```

```
## Warning: geom_vline(): Ignoring `mapping` because `xintercept` was provided.
```

```
## Warning: geom_vline(): Ignoring `data` because `xintercept` was provided.
```

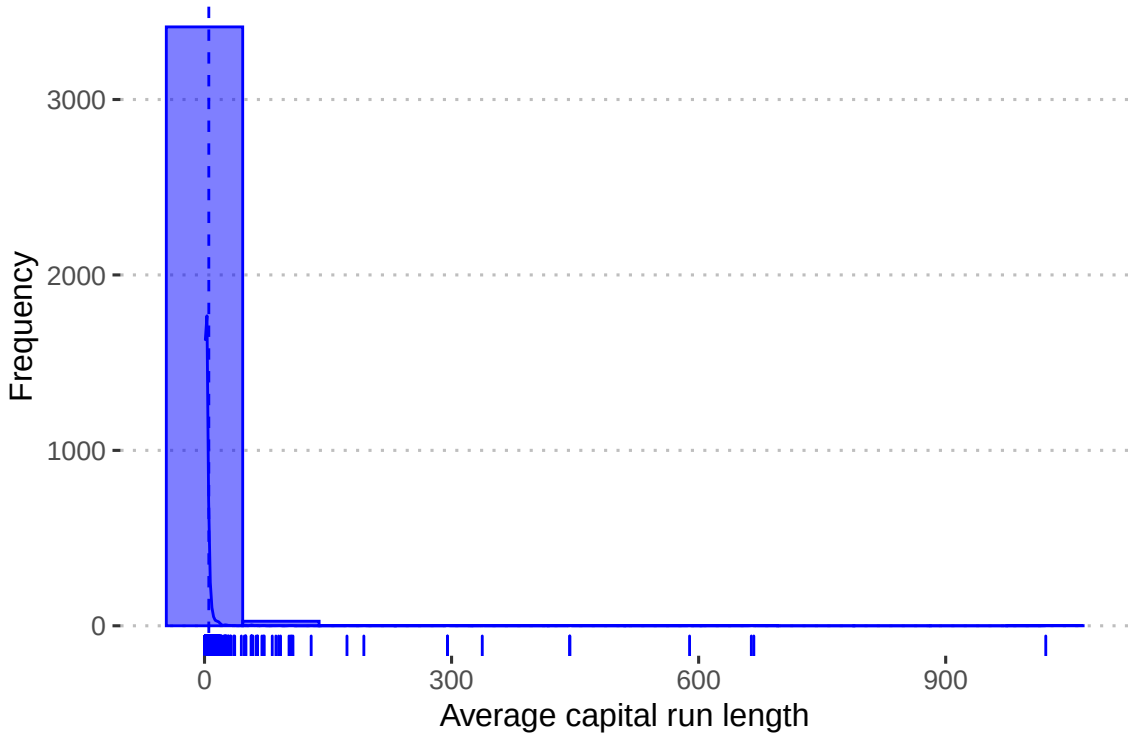

Figure 11: Histogram of the `capitalAve` variable. The distribution is far from normal.

For the `capitalAve` variable, for example, the mean is 5.21, but the standard deviation is much larger: 30.23.

In this case, it is important to pro-process this variable, so that the machine learning algorithm does not get tricked by the fact the variable is skewed and highly variable.

5.2 Standardising

One option is to transform into Z scores. Centring and scaling is one approach, and that will take care of some the problems that we see in these data.E.g.:

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

Now, obviously the mean is 0, and the standard deviation is 1.

5.2.1 Standardising - test set

Now, when we applied the same standardisation to the testing data set, we do it **based on the mean and SD of the training set**.

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

Because of this, mean will not equal 0, and SD will not equal 1, but hopefully these values will not be far off. In fact, when doing this for the testing set, the mean is -0.0029664, and the standard deviation is 1.19.

5.2.2 Standardising - preProcess function

Alternatively, `caret` has a `preProcess` function. Here, we pass all the predictor variables from the data set (except fro the number 58, which is the outcome variable we are trying to predict).

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

Here, again, the mean is 0, and the standard deviation is 1.

We can also apply this to the test set:

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

Here, the mean is -0.0029664, and the standard deviation is 1.19.

5.2.3 Standardising - preProcess as argument (in the train function)

`preProcess` can also be passed as an argument to the `train` function.

```
RNGkind(sample.kind = "Rounding") #to make seed equivalent to older R versions
set.seed(32343)

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

```
## Generalized Linear Model
##
## 3451 samples
##   57 predictor
##    2 classes: 'nonspam', 'spam'
##
## Pre-processing: centered (57), scaled (57)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 3451, 3451, 3451, 3451, 3451, 3451, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9149232  0.8192591
```

5.3 Standardising - Box-Cox transforms

Alternatively, you can use other transformations, like Box-Cox transforms. These are 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.

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

p1 <- gghistogram(cbind(training,trainCapAveS), x = "trainCapAveS",
```

```
            fill = "red",
            color = "red",
            add = "mean",
            rug = TRUE,
            bins = 17,
            palette = "blue") +
  labs(x = "Average capital run length",
       y = "Frequency") +
  theme_pubclean()
p2 <- ggqqplot(cbind(training, trainCapAveS), x = "trainCapAveS",
               color = "red",
               ggtheme = theme_pubclean())

ggarrange(p1, p2,
          labels = "AUTO",
          ncol = 2)
```

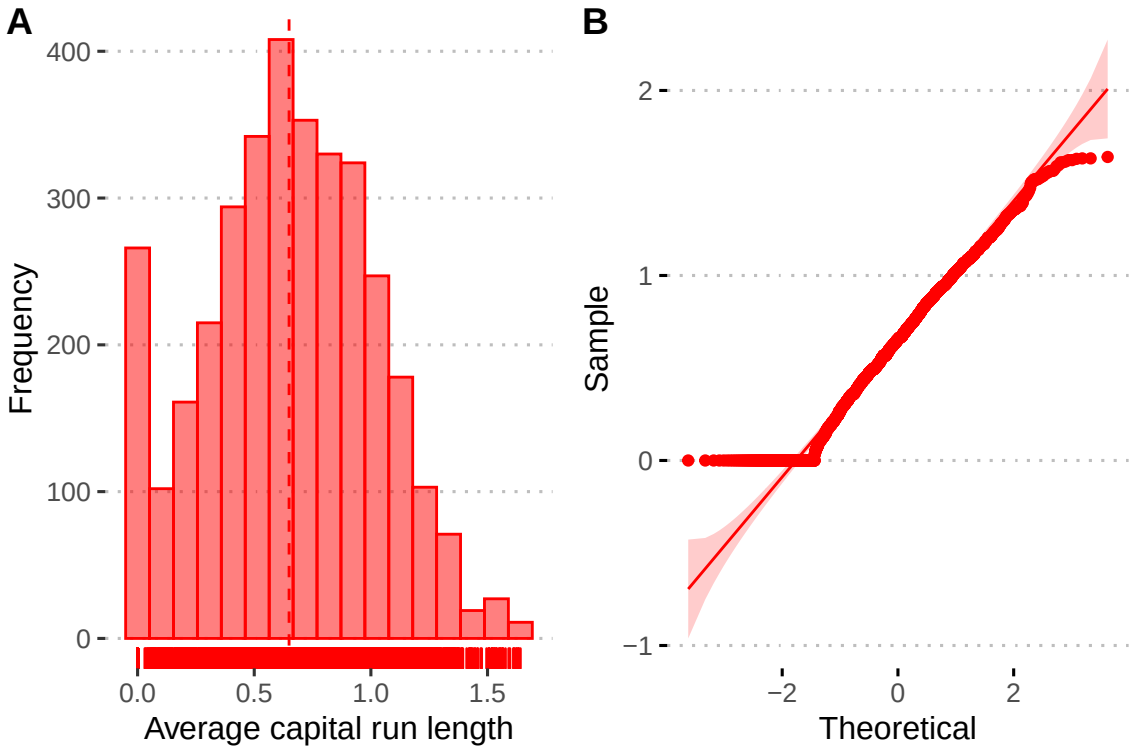

Figure 12: Distribution of the Box-Cox transformed `trainCapAveS` variable. **A.** Histogram. **B.** Q-Q plot.

In the Q-Q plot is clear how values that are repeated cannot be *fixed* with that transformation (in this case, a bunch of zeroes).

5.4 Standardising - Imputing data

It is obviously common to have missing data. In such cases, prediction algorithms often fail. We can use `method = "knnImpute"` in the `preProcess` function. `"knnImpute"` refers to **k-nearest neighbours (KNN)** imputation, a simple machine learning algorithm that computes the k. So if k equal to ten, then it takes the 10 nearest, data vectors that look most like data vector with the missing value, and average the values of the variable that is missing and compute them at that position.

Here, we do that, and then standardise the values (to Z scores).

```
RNGkind(sample.kind = "Rounding") #to make seed equivalent to older R versions
set.seed(13343)

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

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

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

We can now compare the quantiles of the true values, versus the imputed ones

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

```
##            0%           25%           50%           75%          100%
## -1.6031757187 -0.0006280135  0.0006405772  0.0012662413  0.0890718906
```

We can also check only the values that were imputed (`NAs`), and compare them to the values that were not `NA`. Here, we can see that values that were imputed are a little bit farther appart from 0, compare to the ones that were not, but not too much.

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

```
##           0%          25%          50%          75%         100%
## -1.603175719 -0.020333881  0.001051511  0.016187791  0.089071891
```

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

```
##            0%           25%           50%           75%          100%
## -0.8576155730 -0.0005543317  0.0006384720  0.0012283478  0.0017251739
```

5.5 Notes and further reading

- Training and test sets must be **processed in the same way:** The `caret` package handles a lot of this under the hood in the sense that if you train a data set using `preProcess` functions, built into the `train` function, it applies that preprocessed function to the test set. It will handle all of the preprocessing in the correct way for you
- **Test transformation will likely be imperfect**
  - Especially if the test/training sets were collected at different times
  - But, you need to make sure any transformation you apply to the training set, is done in the same way to the test set
- Be **careful when transforming factor variables!** It is much more difficult to know how what is the right transformation. Most machine learning algorithms are built to deal with either:
  - binary predictors (in which case the binary predictors are not pre-processed), or

  - continuous predictors in which case sometimes it's expected that, the data are preprocessed to look more normal

- preprocessing with caret

---

6 Lecture 6: Covariate creation

This lecture is about covariate creation. Covariates are sometimes called predictors and sometimes called features. They're the variables that you will actually include in your model that you're going to be using to combine them to predict whatever outcome that you care about.

There are two levels of covariate creation, or feature creation:

1. The first level is, taking the raw data that you have and turning it into a predictor that you can use. So the raw data often takes the form of an image, or a text file, or a website. That kind of information is very hard to build a predictive model around when you have not summarised the information in some useful way into either a quantitative or qualitative variable
   - what 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 some compression and making it easier to fit standard machine-learning algorithms
   - in 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
2. Transforming tidy covariates

```
knitr::include_graphics("featlevels.png")
```

Figure 13: The two levels of covariate (or feature) creation, with the example of an email.

First level: In this example of an email, first, the average number of capitals (actually, the proportion) that are in the email (in this case 100% of the letters in the email are capital letters); second, frequency a particular word appears (for example, how often does *you* appear?); third, number of dollar signs (this might be a really good predictor of whether an email is spam or not). Here you can see there are a large number of dollar signs (8), so we calculated another feature of that data set.

Second level: it might not be the average number what is 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. So, the next stage is transforming the variables into more useful variables.

6.1 Level 1: Raw data <- covariates

- Depends heavily on application
- The balancing act is summarisation vs. information loss
  - 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.
- Examples:
  - Text files: frequency of words, frequency of phrases (a good example is Google ngrams),),) frequency of capital letters
  - Images: Edges, corners, blobs, ridges (computer vision feature detection)
  - Webpages: Number and type of images, position of elements, colours, videos (A/B Testing)
  - People: Height, weight, hair colour, 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 creating more features
  - it is **better to create more features, and then filter them out** in the model-building process
- Can be automated, but use caution!
  - 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; it won't generalize well

6.2 Level 2: Tidy covariates <- new covariates

The second level is taking tidy covariates (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.

- More necessary for some methods (regression, svms) than for others (classification trees).
- Should be done ***only on the training set***
- The best approach is through exploratory analysis (plotting/tables)
- When using the `caret` package, these **new covariates should be added to data frames**

6.3 Example

6.3.1 Load data

```
library(ISLR)
library(caret)

data(Wage)

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

6.3.2 Common covariates to add, dummy variables

Basic idea: convert factor variables to indicator variables.

In this case let's look in the training set at the variable called job class. So that job class has two different levels, it's either industrial, or it's information. So one thing that we could try to do is try to plug that variable directly into a prediction model, but the values of that variable will be a actually a set of characters. It'll either be "industrial," or it'll be "information."

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

```
##
##  1. Industrial 2. Information
##           1051           1051
```

And it's sometimes hard for prediction algorithms to use those qualitative information variables, in order to actually do the prediction. So one thing we might want to do is turn it into a quantitative variable, and the way that you can do that with the `caret` package is with this `dummyVars` function. So basically it says we're going to pass in a model so the outcome is wage. Job class is going to be the predictor variable, and the training set is the set where we're going to be building those dummy variables. And then if you predict, using the `predict` function, this dummy's object and a new data set, in this case we're just going to apply it to the training data set, you get, two new variables out.

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

```
##        jobclass.1. Industrial jobclass.2. Information
## 86582                       0                       1
## 161300                      1                       0
## 155159                      0                       1
## 11443                       0                       1
## 376662                      0                       1
## 450601                      1                       0
```

So the first is an indicator that you are *industrial*, and the second is an indicator that you're *information*. If the indicator of industrial is 1, it means that for that person has an industrial job. If it's 0, it means for that person, they had not an industrial job. So the same thing is true for information.

So, in this case, where's there only two different levels of this variable, there's only industrial and information, then whenever you're 1 for industrial, you're 0 for information, and whenever you're 0 for industrial, you're 1 for information and so forth.

But if you had three variables here, it would probably have, 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.

6.3.3 Removing zero covariates

Another thing that happens is that some of the variables are basically have no variability in them.

So it's often that you'll create a feature for example, if you create a feature that says for emails, does it have any letters in it at all? Almost every single email will have lots, have at least one letter in it, so that variable will always be equal to true. It's always got letters in it, so it has no variability and it's probably not going to be a useful covariate.

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

```
##            freqRatio percentUnique zeroVar   nzv
## year        1.037356    0.33301618   FALSE FALSE
```

```
## age         1.027027    2.85442436   FALSE FALSE
## maritl      3.272931    0.23786870   FALSE FALSE
## race        8.938776    0.19029496   FALSE FALSE
## education   1.389002    0.23786870   FALSE FALSE
## region      0.000000    0.04757374    TRUE  TRUE
## jobclass    1.000000    0.09514748   FALSE FALSE
## health      2.468647    0.09514748   FALSE FALSE
## health_ins  2.352472    0.09514748   FALSE FALSE
## logwage     1.061728   19.17221694   FALSE FALSE
## wage        1.061728   19.17221694   FALSE FALSE
```

For example, here we can see that it tells us the percentage of unique values for a particular variable, so in this case the variable has about 0.33% unique values, and it's not near zero variance variable, but for example, the variable sex, only is basically males and so it has a very low frequency ratio.

In other words, it's basically all one category, and so, this ends up being a near zero variable and so, you could use this column of the matrix to throw out all those variables like sex and, in this case, region, that are variables that don't really have any variability in them and shouldn't be used in prediction algorithms. So this is a nice way to throw those less meaningful predictors out right away.

6.4 Spline basis

If you do linear regression or generalized linear regression as your prediction algorithm (future lecture), the idea will be to fit, basically straight lines through the data.

Sometimes, you want to be able to fit curvy lines, and one way to do that is with a `basis` functions, and so you can find those, for example, in the `splines` package.

The `bs` function will create a polynomial variable. So in this case, we pass at a single variable, in this case, the training set, we take the *Age* variable, and we say we want a third degree polynomial for this variable (`df = 3`).

```
library(splines)

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

```
##              1          2           3
## [1,] 0.2368501 0.02537679 0.000906314
## [2,] 0.4163380 0.32117502 0.082587862
## [3,] 0.4308138 0.29109043 0.065560908
## [4,] 0.3625256 0.38669397 0.137491189
## [5,] 0.3063341 0.42415495 0.195763821
## [6,] 0.4241549 0.30633413 0.073747105
```

So when you do that, you essentially get, you'll get a three-column matrix out. So this is now three new variables:

1. The first variable corresponds to age, the actual *Age* values (but scaled for computational purposes)
2. Age^2
3. Age^3

If you include these covariates in the model instead of just the age variable when you're fitting a linear regression, you allow for curvy model fitting.

6.5 Fitting curves with splines

We can fit a model polynomial model (`lm1`) with the variables created from *Age*.

```
lm1 <- lm(wage ~ bsBasis,
          data = training)
summary(lm1)
```

```
##
## Call:
## lm(formula = wage ~ bsBasis, data = training)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -100.47  -24.76   -5.05   15.35  199.04
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   55.513      4.966  11.178  < 2e-16 ***
## bsBasis1     108.579     13.963   7.776 1.16e-14 ***
## bsBasis2      55.068      9.881   5.573 2.82e-08 ***
## bsBasis3      34.054     14.109   2.414   0.0159 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 39.29 on 2098 degrees of freedom
## Multiple R-squared:  0.09261,    Adjusted R-squared:  0.09132
## F-statistic: 71.38 on 3 and 2098 DF,  p-value: < 2.2e-16
```

And plot this model:

```
#In base R
#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 ggplot2
ggplot(training, aes(x = training$age, y = training$wage)) +
  geom_point(alpha = 0.3) +
  geom_line(aes(y = predict(lm1, newdata = training)),
            color = "red")  +
  labs(x = "Age", y = "Wage") +
  theme_pubclean()
```

```
## Warning: Use of ‘training$age‘ is discouraged. Use ‘age‘ instead.
```

```
## Warning: Use of ‘training$wage‘ is discouraged. Use ‘wage‘ instead.
```

```
## Warning: Use of ‘training$age‘ is discouraged. Use ‘age‘ instead.
```

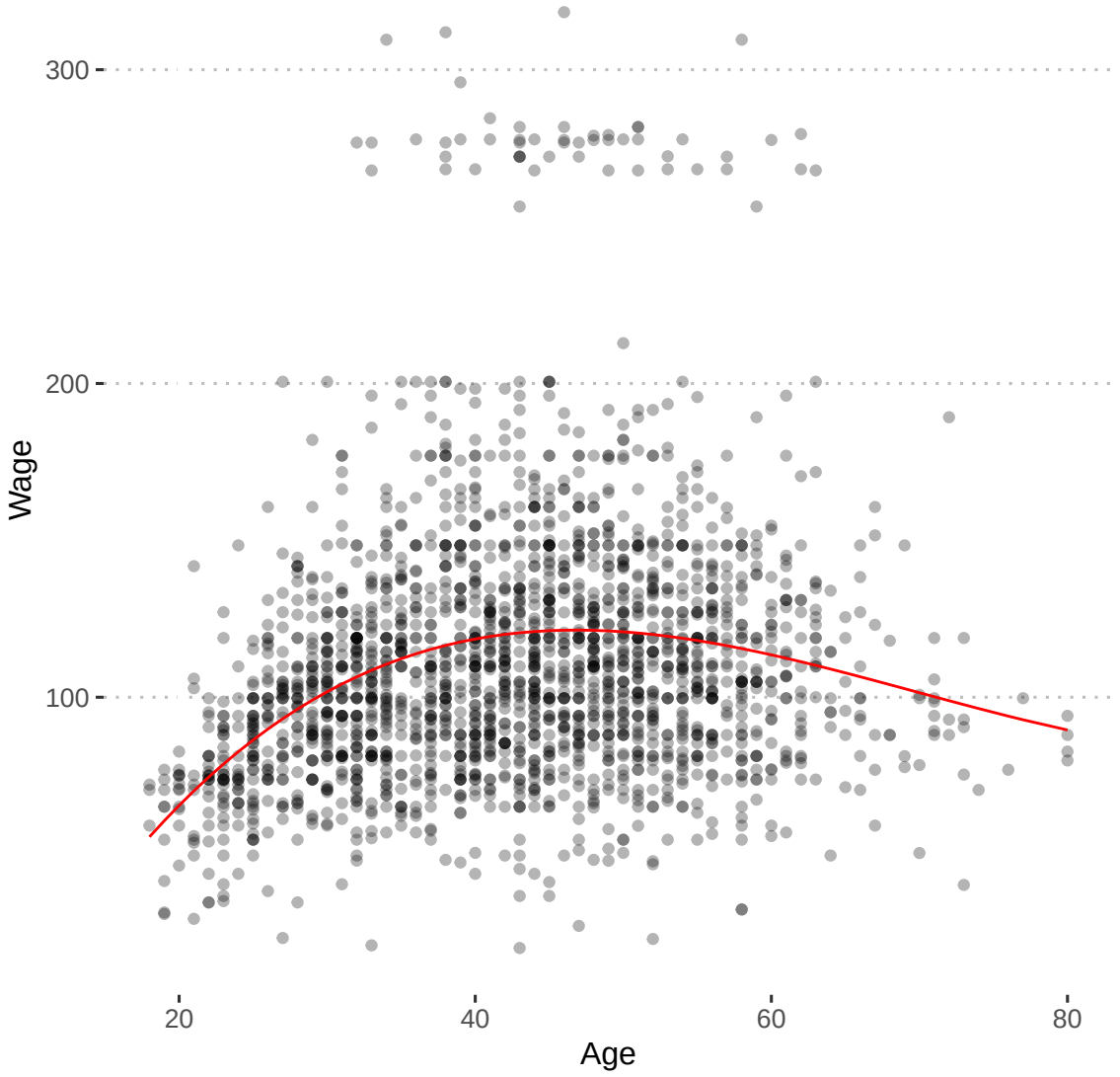

Figure 14: Model to predict Wage from Age. In this case, we used a model with the polynomial model: Age, Age^2, and Age^3 as predictors. The red line represents the model.

6.5.1 Splines on the test set

Then on the test set, you'll have to predict those same variables.

In this example, we predict Age in the testing set, using the exact same procedure used on the training set. So you can do that by saying I'm going to predict from this variable that I created using the BS function:

```
bsBasisAgeTesting <- predict(bsBasis,
                             age = testing$age)
head(bsBasisAgeTesting)
```

```
##              1          2           3
## [1,] 0.2368501 0.02537679 0.000906314
## [2,] 0.4163380 0.32117502 0.082587862
## [3,] 0.4308138 0.29109043 0.065560908
## [4,] 0.3625256 0.38669397 0.137491189
## [5,] 0.3063341 0.42415495 0.195763821
```

```
## [6,] 0.4241549 0.30633413 0.073747105
```

6.6 Notes and further reading

- Level 1 feature creation (raw data to covariates)
  - Science is key. Google "feature extraction for [data type]"
  - Err on overcreation of features
  - 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 track.

---

7 Lecture 7: Preprocessing with PCA

This is a lecture a lecture about preprocessing covariants with principal components analysis.

You have multiple quantitative variables that sometimes are highly correlated with each other. In other words, they are 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.

7.1 Correlated predictors

Here, we are looking at the absolute values (`abs`) of the correlations between all variables (excluding the dependant variable), and selecting those that are > 0.8.

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

In the slides, they only get 1 large correlation (between `num415`, and `num857`; 2 rows). I, however, get 3:

- `num415` and `num857`
- `direct` and `num857`
- `direct` and `num415`

```
p1 <- ggplot(spam, aes(x = num415, y = num857)) +
  geom_point(alpha = 0.3, color = "red") +
  theme_pubclean()
p2 <- ggplot(spam, aes(x = direct, y = num857)) +
  geom_point(alpha = 0.3, color = "purple") +
  theme_pubclean()
p3 <- ggplot(spam, aes(x = direct, y = num415)) +
  geom_point(alpha = 0.3, color = "blue") +
  theme_pubclean()

ggarrange(p1, p2, p3,
          ncol = 3,
          labels = "AUTO")
```

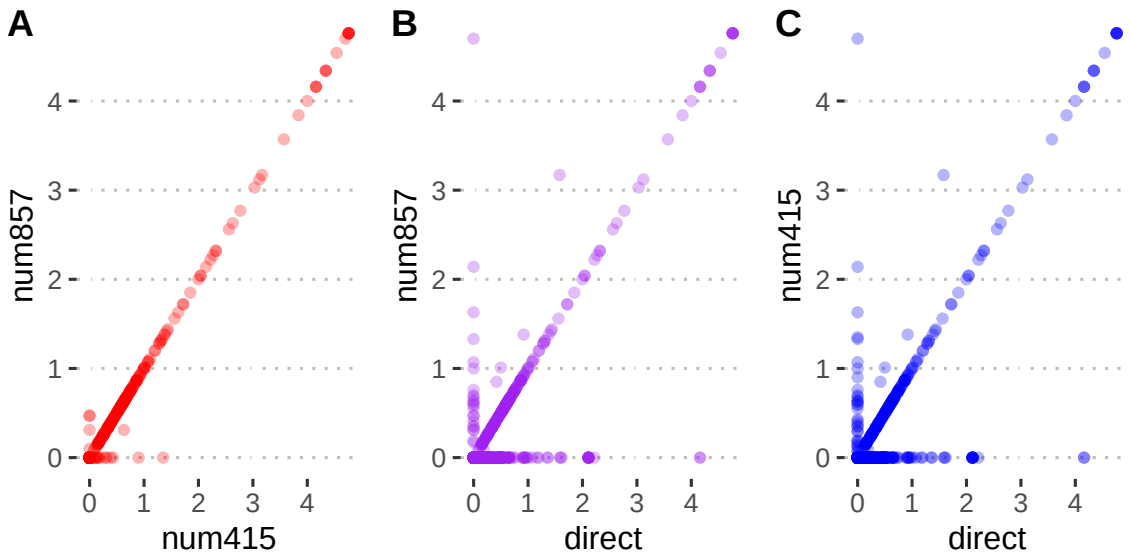

Figure 15: Correlations with $r > 0.8$ between predictor variables in the `spam` database. In the lectures, only panel **A** appears, but I found these 3 correlations between the same 3 variables.

7.2 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)

7.3 We could rotate the plot

So just as an example, here's a combination I could do. I could say I could take 0.71 (check section 7.6) times the 415 variable plus 0.71 times 857 variable. And create a new variable called x. Which is basically the sum of those two variables. Then I could take the difference of those two variables. By basically doing 0.71 times 415 minus 0.71 times 857. So this is basically adding, x is adding the two variables together, y is subtracting the two variables.

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

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

```
X <- 0.71*training$num415 + 0.71*training$num857
Y <- 0.71*training$num415 - 0.71*training$num857

ggplot(data.frame(cbind(X, Y)), aes(x = X, y = Y))  +
  geom_point(alpha = 0.3, color = "purple") +
  theme_pubclean()
```

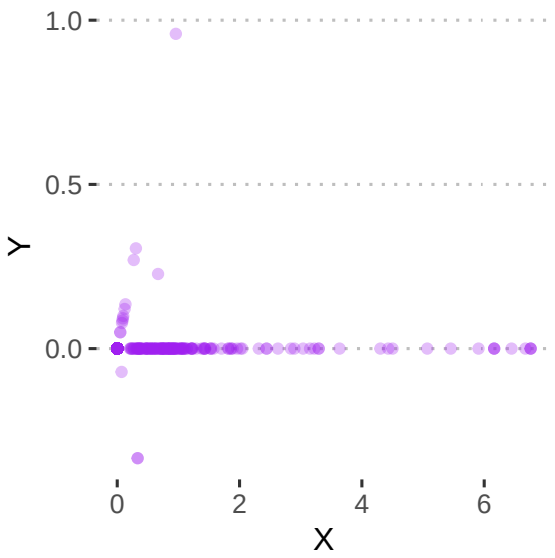

Figure 16: Rotated correlations.

So then if I plot those variables versus each other, when I add them up, that's the x-axis, and when I take the difference, that's the y-axis. And so you can see most of the variability is happening in the x-axis.

In other words there's lots of points all spread out across the x-axis, but most of the points are clustered right here at 0 on the y-axis. So that almost all of these points have a y value of 0.

So, the adding the two variables together captures most of the information in those two variables and subtracting the variables takes less information. So the idea here is we might want to use the sum of the two variables as a predictor.

7.4 Related problems

You have multivariate variables $X_1, \ldots, X_n$ so $X_1 = (X_{11}, \ldots, X_{1m})$

- Find a new set of multivariate variables that are uncorrelated and explain as much variance 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.
- If you put all the variables together in one matrix, find the best matrix created with fewer variables (lower rank) that explains the original data.

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

7.5 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 the SVD is a "matrix decomposition"

$$X = UDV^T$$

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

PCA

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

7.6 PCA in R - `prcomp`

PCA between `num415`and `num857`.

```
smallSpam <- spam[,c(34,32)]
prComp <- prcomp(smallSpam)

ggplot(data.frame(prComp$x), aes(x = PC1, y = PC2))  +
  geom_point(alpha = 0.3, color = "red") +
  theme_pubclean()
```

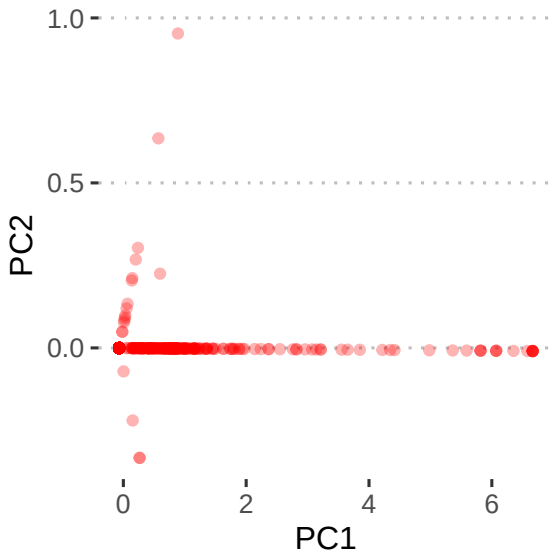

Figure 17: PCA between `num415` and `num857`.

You can check the rotation as well (hence the 0.71 used in section 7.3.

```
round(prComp$rotation, 4)
```

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

PC1 is 0.7081 × `num415`, and 0.7061 × `num857`. The component that explains the most variance (PC1) is adding the 2 variables, and PC2 is the subtraction of these.

7.7 PCA on SPAM data

I've applied a function of the data set, the log_{10} transform, and added 1. I've done this to make the data look a little bit more Gaussian (some of the variables are skewed).

```
prComp <- prcomp(log10(spam[,-58] + 1))

ggplot(data.frame(prComp$x), aes(x = PC1, y = PC2, color = spam$type))  +
  geom_point(alpha = 0.3) +
  labs(color = "Spam type") +
  theme_pubclean()
```

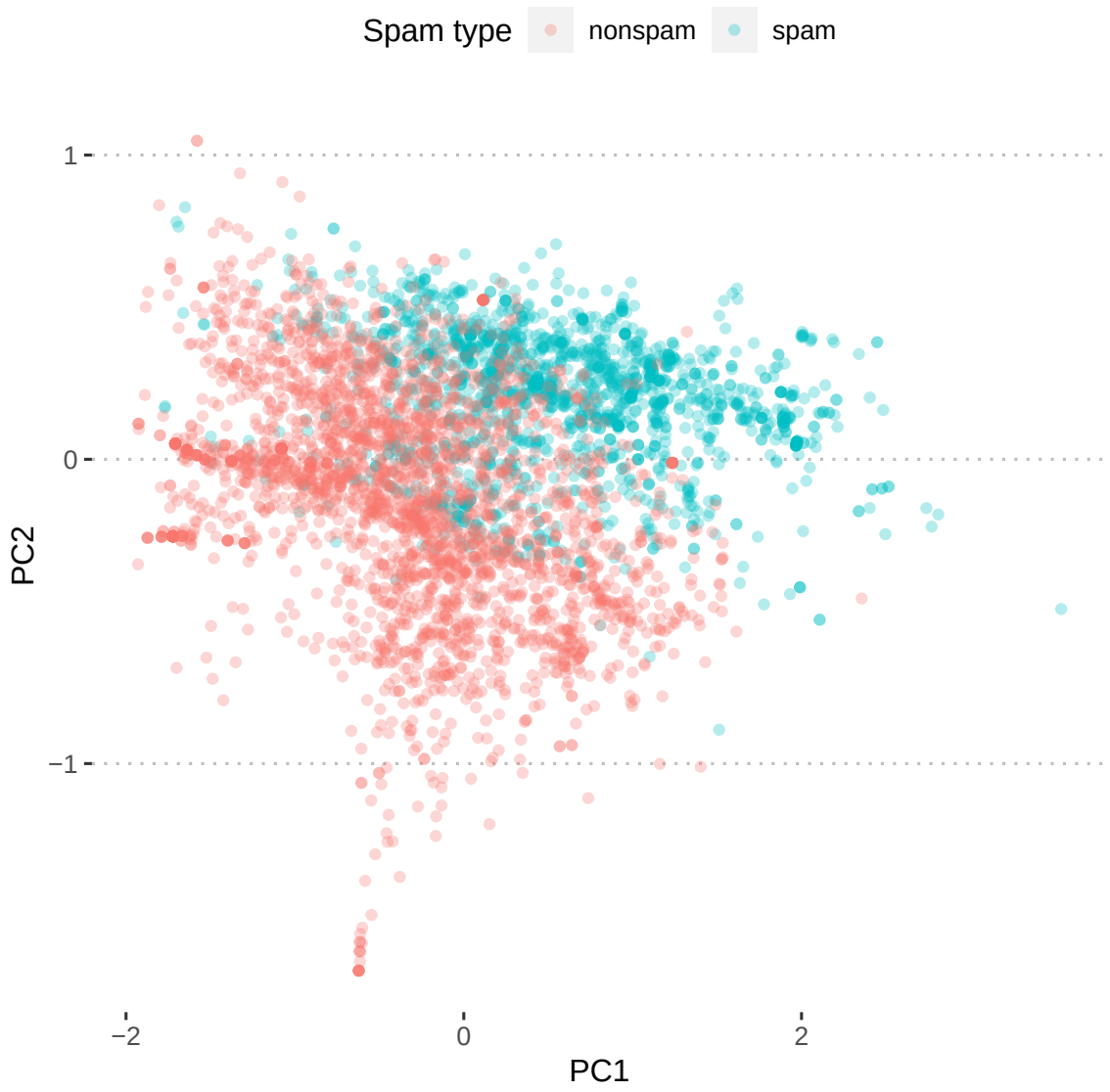

Figure 18: PCA on the SPAM data using `prcomp`.

7.8 PCA with caret

PCA can be achieved with `caret`, by using the `method = "pca"` argument in the `preProcess` function. To obtain the PCA scores, the `predict` function must be used.

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

spamPC <- predict(preProc,log10(spam[,-58]+1))

ggplot(spamPC, aes(x = PC1, y = PC2, color = spam$type))  +
  geom_point(alpha = 0.3) +
  labs(color = "Spam type") +
  theme_pubclean()
```

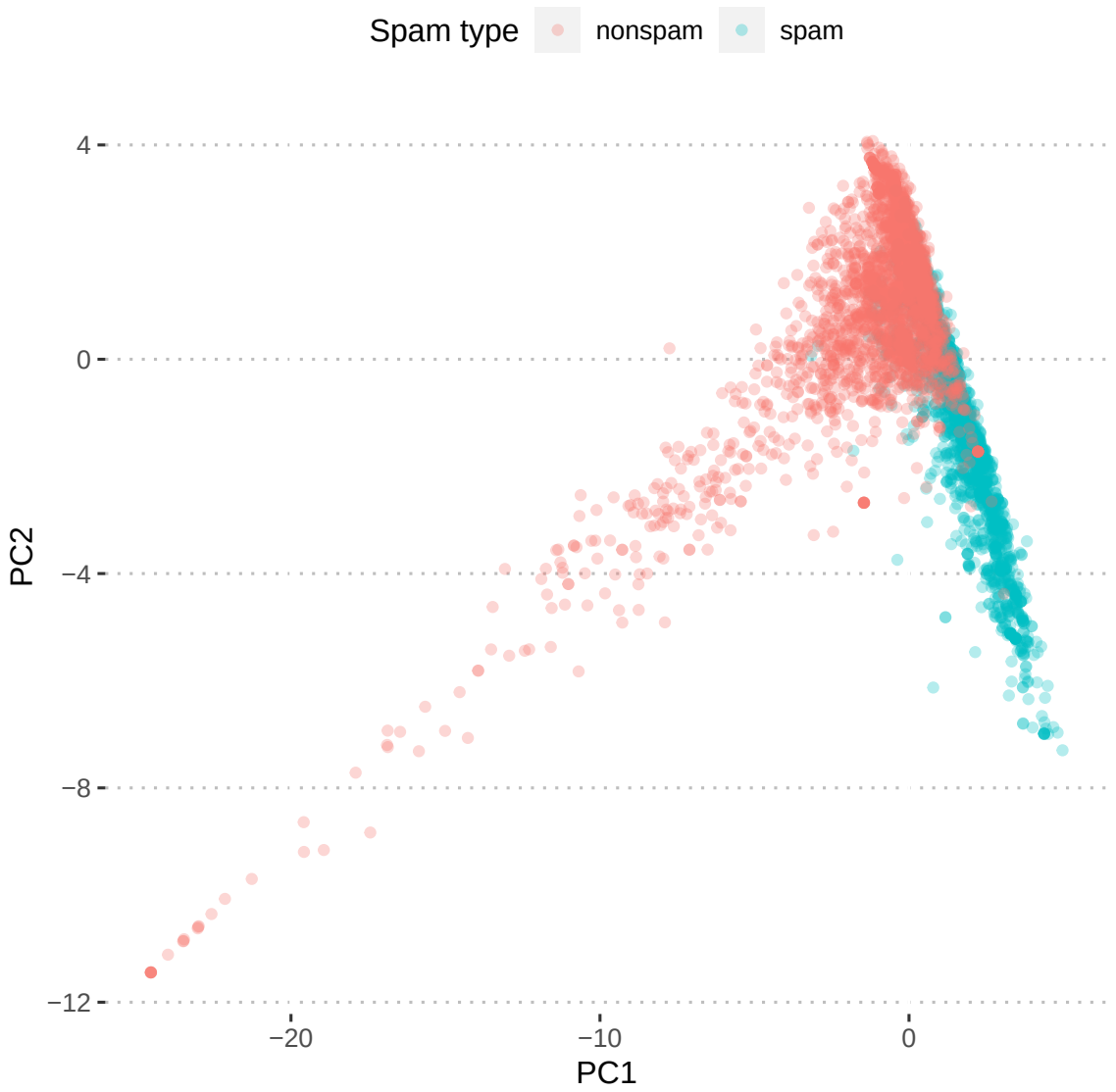

Figure 19: PCA on the SPAM data using `caret`.

7.8.1 Preprocessing with PCA

Then, you can use the PCA scores to `train` a model:

```
preProc <- preProcess(log10(training[,-58]+1),
                      method = "pca",
                      pcaComp = 2)

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

# The normal syntax doesn't work
#modelFit <- train(training$type ~.,
#                  method="glm",
#                  data=trainPC)

# This syntax works
modelFit <- train(x = trainPC, y = training$type, method="glm")
```

Obviously, in the test data set, you must use the same principal components calculated in the training data set. To do this, we first `predict` PCA scores on the testing data set, using the `preProc` object created earlier. Then, we can create the `confusionMatrix` for the testing set, using the `modelFit` model, and the PCA scores on the `testPC` object.

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

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction nonspam spam
##    nonspam     645   52
##    spam         73  380
##
##                Accuracy : 0.8913
##                  95% CI : (0.8719, 0.9087)
##     No Information Rate : 0.6243
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.7705
##
##  Mcnemar's Test P-Value : 0.07364
##
##             Sensitivity : 0.8983
##             Specificity : 0.8796
##          Pos Pred Value : 0.9254
##          Neg Pred Value : 0.8389
##              Prevalence : 0.6243
##          Detection Rate : 0.5609
##    Detection Prevalence : 0.6061
##       Balanced Accuracy : 0.8890
##
##        'Positive' Class : nonspam
##
```

Here, we calculated a relatively small number of principal components, but still have a relatively high accuracy in prediction. So principal component analysis can reduce the number of variables while maintaining accuracy.

7.9 Alternative (sets # of PCs)

We can also add the PCA straight into the `train` function, in the `preProcess` argument.

```
# The normal syntax doesn't work
#modelFit <- train(training$type ~.,
#                  method = "glm",
#                  preProcess = "pca",
#                  data = training)

# This syntax works
```

```
modelFit <- train(x = training[,-58], y = training$type,
                  method="glm",
                  preProcess = "pca")

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction nonspam spam
##    nonspam     658   39
##    spam         50  403
##
##                Accuracy : 0.9226
##                  95% CI : (0.9056, 0.9374)
##     No Information Rate : 0.6157
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.8372
##
##  Mcnemar's Test P-Value : 0.2891
##
##             Sensitivity : 0.9294
##             Specificity : 0.9118
##          Pos Pred Value : 0.9440
##          Neg Pred Value : 0.8896
##              Prevalence : 0.6157
##          Detection Rate : 0.5722
##    Detection Prevalence : 0.6061
##       Balanced Accuracy : 0.9206
##
##        'Positive' Class : nonspam
##
```

7.10 Final thoughts on PCs

- 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 (Hastie et al., 2009)

8 Lecture 8: Predicting with regression

This lecture's about one of the most direct and simple ways to perform machine learning using **regression modelling**.

8.1 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

8.2 Example: Old faithful eruptions

This example is based on the Old Faithful Geyser Data (aka `faithful` database), that only contains 2 variables.

```
library(caret)

data(faithful)

RNGkind(sample.kind = "Rounding") #to make seed equivalent to older R versions
set.seed(333)

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

trainFaith <- faithful[inTrain,]
testFaith <- faithful[-inTrain,]

head(trainFaith)
```

```
##   eruptions waiting
## 1     3.600      79
## 3     3.333      74
## 5     4.533      85
## 6     2.883      55
## 7     4.700      88
## 8     3.600      85
```

As a plot:

```
ggplot(trainFaith, aes(x = waiting, y = eruptions))  +
  geom_point(alpha = 0.3,
             color = "darkgreen") +
  labs(x = "Waiting Time (m)", y = " Eruption Duration (m)") +
  theme_pubclean()
```

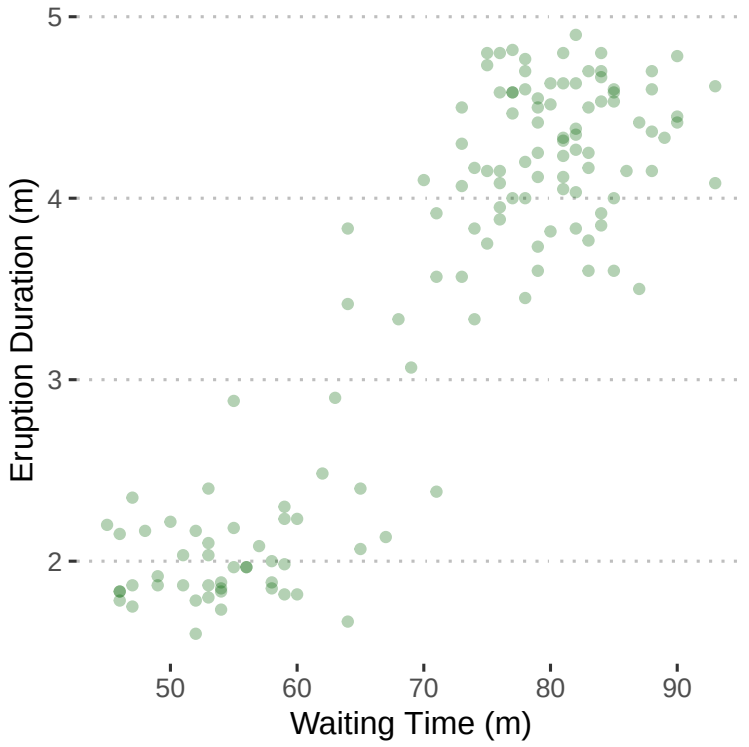

Figure 20: Association between `eruptions` and `wating`.

8.2.1 Fit a linear model

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

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

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

The model is:

```
ggplot(trainFaith, aes(x = waiting, y = eruptions))  +
  geom_line(aes(y = predict(lm1)),
            color = "darkgreen") +
  geom_point(alpha = 0.3,
             color = "darkgreen") +
  labs(x = "Waiting Time (m)", y = " Eruption Duration (m)") +
  theme_pubclean()
```

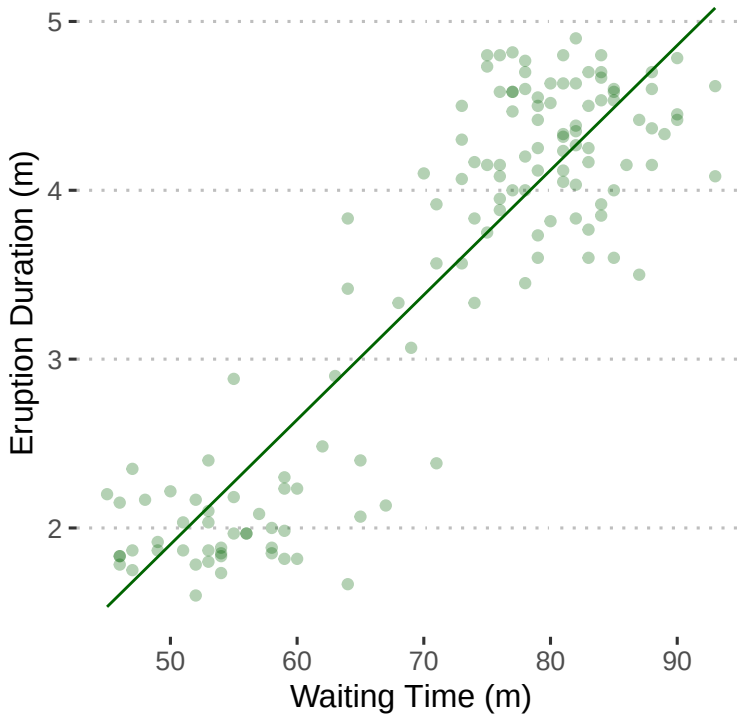

Figure 21: Linear regression between `eruptions` and `wating`.

To predict a value:

$\hat{ED} = \hat{b}_0 + \hat{b}_1 WT$[1]

Here, there is no error term as we do not know what the error is for a particular value (in this example, 80).

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

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

Or

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

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

[1]The hat (^): In simple linear regression with observations of independent variable data x_i and dependent variable data y_i, and assuming a model of $y_i = \beta_0 + \beta_1 x_1 + \varepsilon_i$, can lead to an estimated model of the form $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$ where $\sum_i (y_i - \hat{y}_i)^2$ is minimized via least squares by finding optimal values of $\hat{\beta}_0$ and $\hat{\beta}_1$ for the observed data.

8.2.2 Plot predictions: training and test

```
p1 <- ggplot(trainFaith, aes(x = waiting, y = eruptions)) +
  geom_line(aes(y = predict(lm1)),
            color = "darkgreen") +
  geom_point(alpha = 0.3,
             color = "darkgreen") +
  labs(x = "Waiting Time (m)", y = " Eruption Duration (m)") +
  theme_pubclean()
p2 <- ggplot(testFaith, aes(x = waiting, y = eruptions)) +
  geom_line(aes(y = predict(lm1, newdata = testFaith)),
            color = "red") +
  geom_point(alpha = 0.3,
             color = "red") +
  labs(x = "Waiting Time (m)", y = " Eruption Duration (m)") +
  theme_pubclean()

ggarrange(p1, p2,
          labels = "AUTO")
```

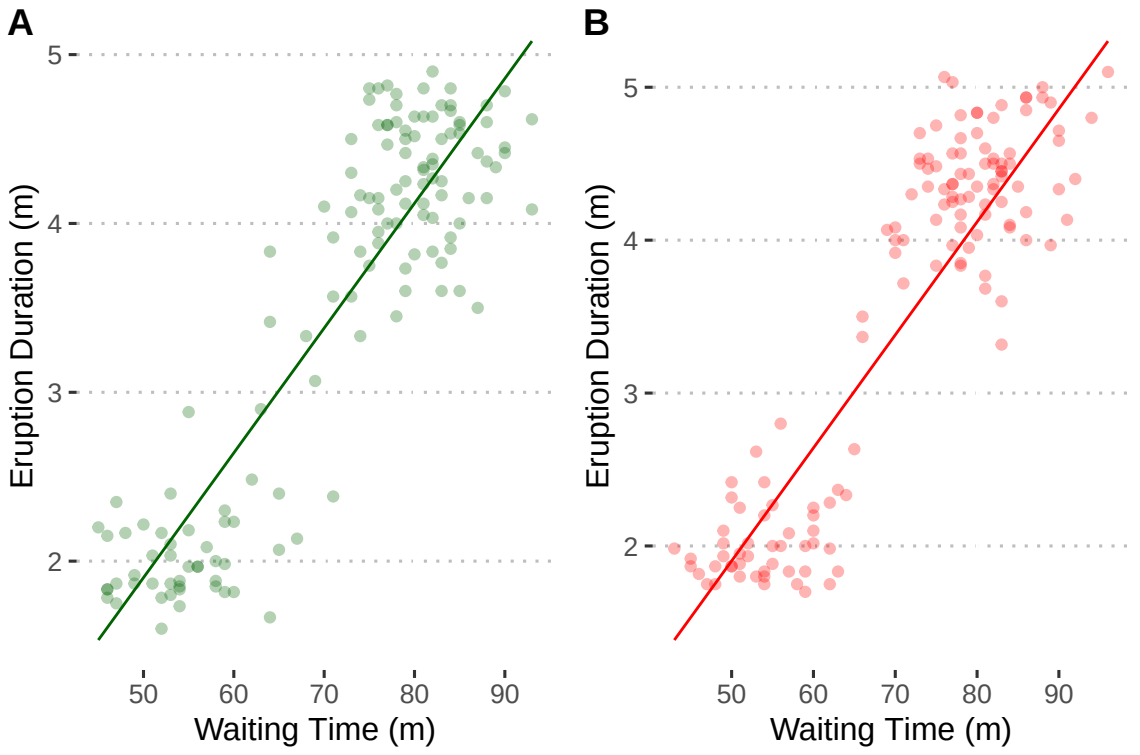

Figure 22: Linear regression between `eruptions` and `wating` from the model applied to the trainig (**A**) and testing (**B**) datasets. On the testing (**B**) dataset, the regression line is a little tilted, but that is expected as the model was not built with these specific data.

8.3 Get training/test set errors

RMSE on the training dataset

```
sqrt(sum((lm1$fitted.values - trainFaith$eruptions)^2))
```

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

RMSE on the test dataset

Obviously, this is a more realistic estimation of the (larger) error that you would get on a new data set compared to the value that we got on the training set.

```
sqrt(sum((predict(lm1, newdata = testFaith) - testFaith$eruptions)^2))
```

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

8.4 Prediction intervals

In base R

```
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)
```

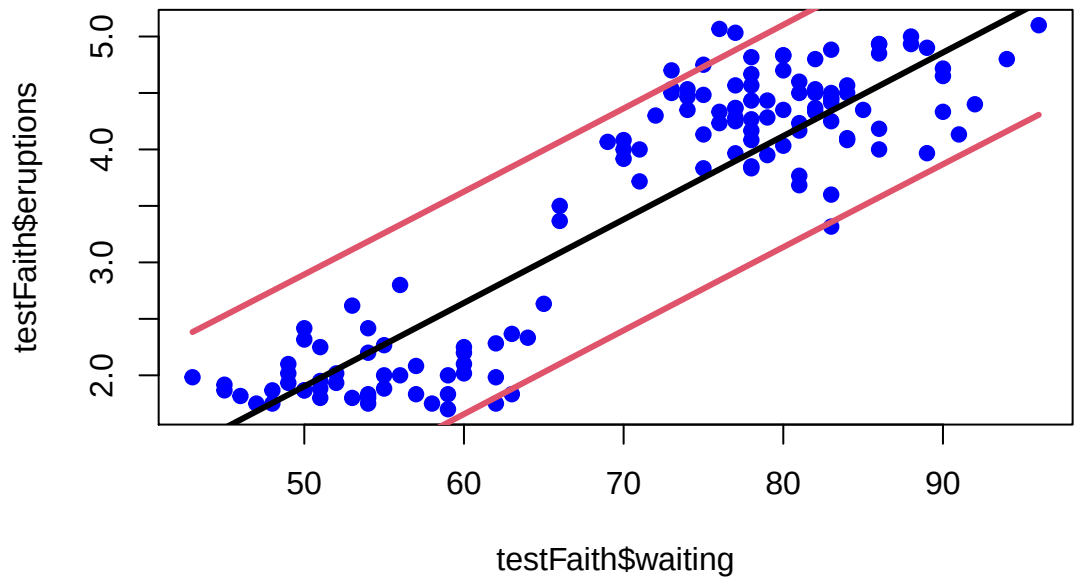

Figure 23: Linear regression between `eruptions` and `wating` from the model applied to testing dataset, with prediction intervals, using base R.

In `ggplot2`

```
pred1 <- data.frame(predict(lm1,
                 newdata = testFaith,
```

```
                interval = "prediction",
                level = 0.95))

ggplot(testFaith, aes(x = waiting, y = eruptions)) +
  geom_line(aes(y = predict(lm1, newdata = testFaith)),
            color = "red") +
  geom_line(aes(y = pred1$lwr),
            color = "red",
            linetype = "dashed") +
  geom_line(aes(y = pred1$upr),
            color = "red",
            linetype = "dashed") +
  geom_point(alpha = 0.3,
             color = "red") +
  labs(x = "Waiting Time (m)", y = " Eruption Duration (m)") +
  theme_pubclean()
```

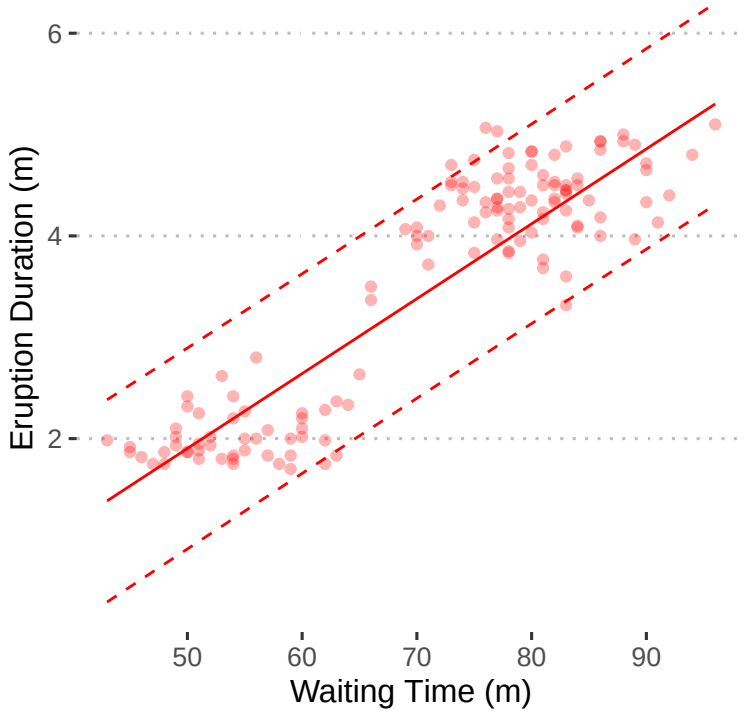

Figure 24: Linear regression between `eruptions` and `wating` from the model applied to testing dataset, with 95% prediction intervals, using `ggplot2`.

8.5 Same process with caret

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

summary(modFit$finalModel)
```

```
##
## Call:
```

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

8.5.1 Comparing predicted to actual values

```
ggplot(testFaith, aes(x = eruptions, y = predict(lm1, newdata = testFaith))) +
  geom_smooth(method = "lm",
              color = "blue") +
  geom_point(alpha = 0.3,
             color = "blue") +
  labs(x = "Actual Eruption Duration (m)",
       y = "Predicted Eruption Duration (m)") +
  stat_cor(aes(label = paste(..rr.label..,
                             cut(..p..,
                                 breaks = c(-Inf,
                                            0.0001,
                                            0.001,
                                            0.01,
                                            0.05,
                                            Inf),
                                 labels = c("'****'",
                                            "'***'",
                                            "'**'",
                                            "'*'",
                                            "' '")),
                             sep = "~")),
           label.y.npc = 0.9,
           color = "black") +
  theme_pubclean()
```

```
## `geom_smooth()` using formula 'y ~ x'
```

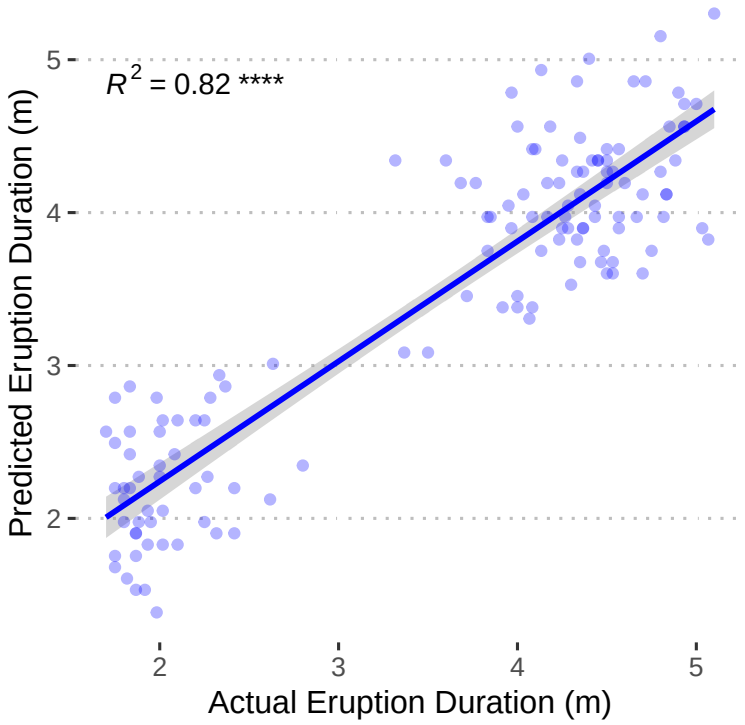

Figure 25: Linear regression between `eruptions` and `wating` from the model applied to testing dataset, with 95% prediction intervals, using `ggplot2`.

8.6 Notes and further reading

- Regression models with multiple covariates can be included
- Often useful in combination with other models
- Elements of statistical learning (Hastie et al., 2009)
- Modern applied statistics with S (Venables & Ripley, 2002)
- Introduction to statistical learning (James et al., 2013)

---

9 Lecture 9: Predicting with Regression Multiple Covariates

This lecture's about two important points:

1. Predicting with regression and using **multiple covariates**
2. 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

9.1 Example: predicting wages

We will again use the `Wage` dataset, from the `ISLR` package, from the book *Introduction to Statistical Learning* (James et al., 2013).

Here, we subset the variable excluding `logwage`, which is the variable we are trying to predict.

```
library(ISLR)
library(ggplot2)
library(caret)
```

```
data(Wage)

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

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

Some important aspect visible from the summary, are that the sample is only of males, and from the Middle Atlantic region.

9.1.1 Get training/test sets

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

training <- Wage[inTrain,]
testing <- Wage[-inTrain,]

dim(training)
```

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

```
dim(testing)
```

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

9.1.2 Feature plot

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

```
            plot = "pairs")
```

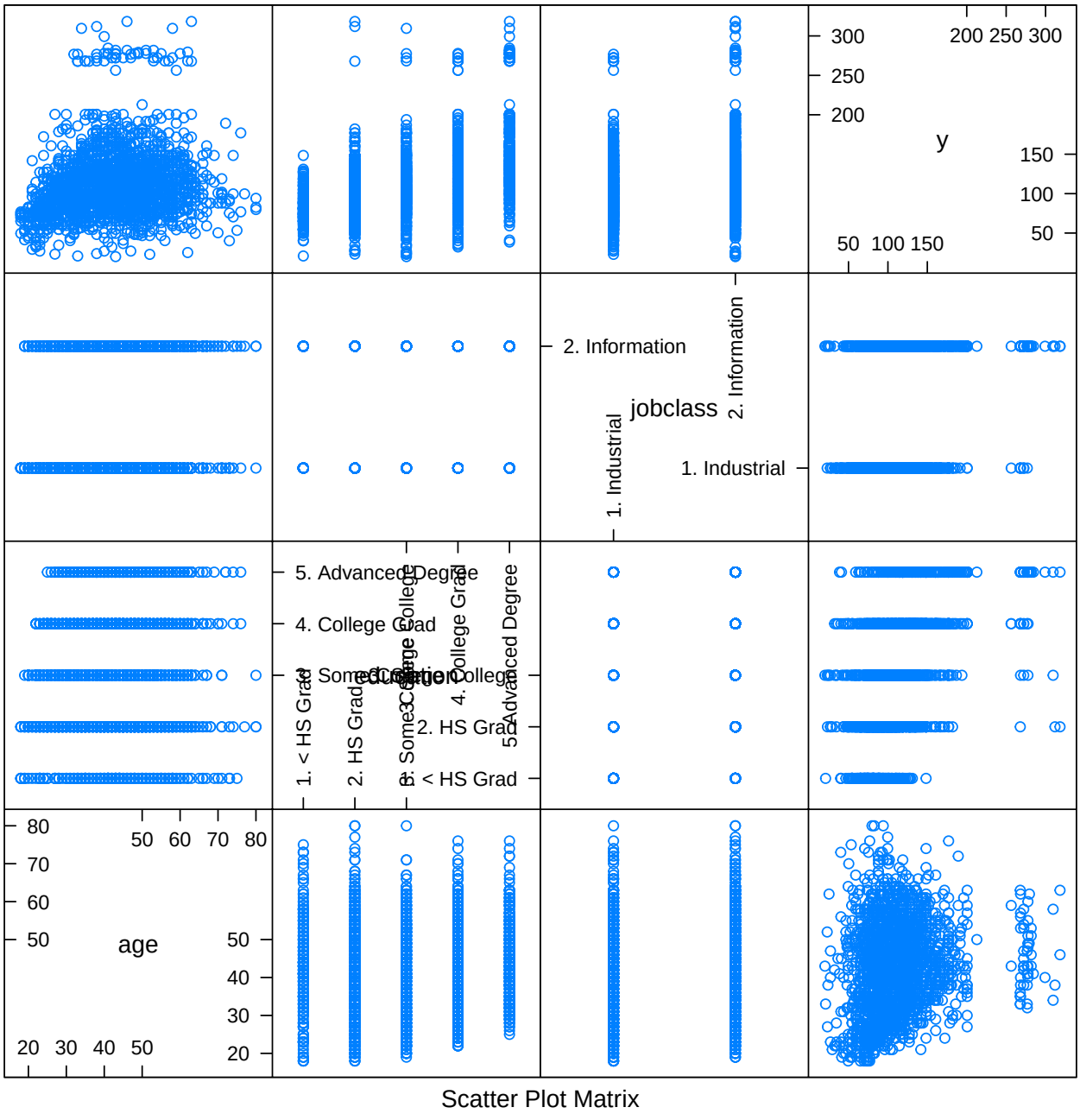

Figure 26: `featurePlot` from the `caret` package. In this case, we asked for the association of three predictors (`age, education, jobclass`) with the dependent variable (`wage`), as well as amongst them. In particular, you need to look at the first row (i.e. the assiociations between the predictors and the outcome variable); in this case, for example, there is a clear association between `ajobclass` and `wage`.

9.1.3 Association between `age` and `wage`

As we saw before, the association between `age` and `wage` is curved, but also there is an important group of outliers, mainly predicted by `jobclass`.

```
p1 <- ggplot(training, aes(x = age, y = wage)) +
  geom_point(alpha = 0.3) +
  labs(x = "Age", y = "Wage") +
  theme_pubclean()

p2 <- ggplot(training, aes(x = age, y = wage, colour = jobclass)) +
  geom_point(alpha = 0.5) +
  labs(x = "Age", y = "Wage", colour = "Job class") +
  theme_pubclean()

p3 <- ggplot(training, aes(x = age, y = wage, colour = education)) +
  geom_point(alpha = 0.5) +
  labs(x = "Age", y = "Wage", colour = "Education") +
  theme_pubclean()

ggarrange(p1, ggarrange(p2, p3,
                        ncol = 2,
                        labels = c("", "C"),
                        align = "h"),
          labels = c("A", "B"),
          nrow = 2)
```

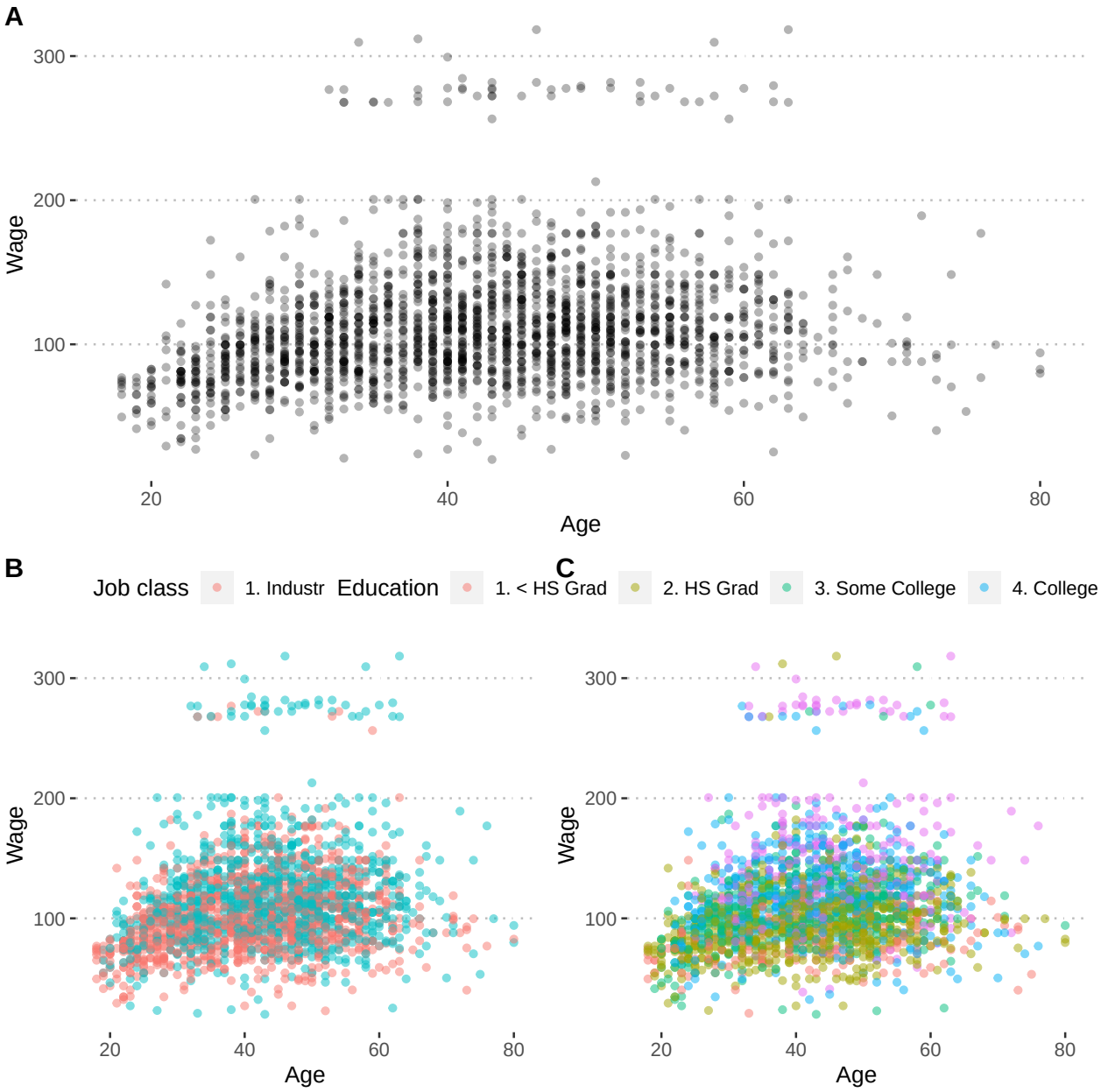

Figure 27: **A.** Association between wage and age. **B.** Association between wage and age by job class. **C.** Association between wage and age by education level.

9.2 Fit a linear model

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

- Here, the idea is that we have some intercept terms (b_0), so that's just the baseline level of wage that we might have
- Then, we might have a relationship with the age of the person ($b_1 age$)
- Then we might have relationship with what job class you're in. So one way that we typically do that

is, by fitting an indicator variable

  - An indicator variable is a variable that's denoted like I in mathematical notation. It just says, if the job class for the i^{th} person is equal to $Information$, this variable's equal to 1. If the job class for the i^{th} person is not equal to information, then this information is equal to 0
  - This represents the difference in the wages between the people with job class equal to $Information$ versus job class equal to not information, when you fix all the other variables in the regression model

- You can also do this for *Education*. This is a little bit more complicated because there are multiple education levels
  - We create an indicator variable for each of the different *Education*
  - And so the variable's equal to 1, if the education for person i^{th} is equal to level K, that variables equal to 1, or otherwise 0

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

finMod <- modFit$finalModel
modFit
```

```
## Linear Regression
##
## 2102 samples
##    3 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 2102, 2102, 2102, 2102, 2102, 2102, ...
## Resampling results:
##
##   RMSE      Rsquared  MAE
##   35.79066  0.248127  24.68782
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

9.3 Diagnostics: Homoscedasticity

Residuals vs fitted values. The line would be centred at zero because the residuals is the difference between the model prediction and the actual real values that we're trying to predict.

Base R

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

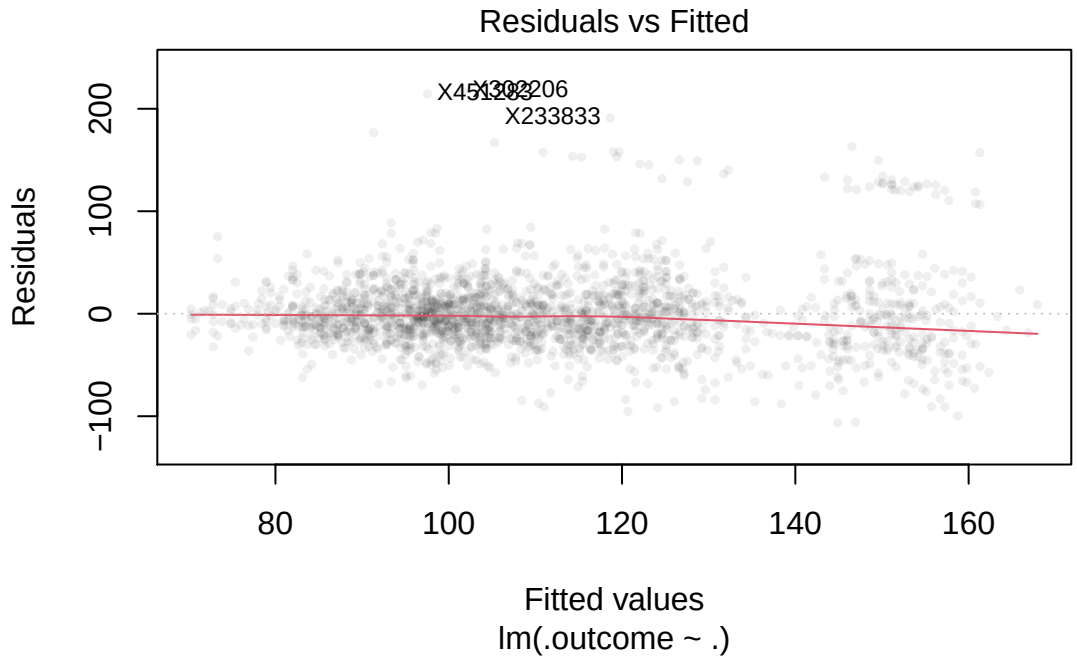

Figure 28: Homoscedasticity of the model with base R. In addition to the red line (which should be around 0), this plor highlights outliers.

In this case, there is still a couple of outliers up here that have been labelled for you in the plot created with Base R.

Performance

```
library(performance)

check_model(finMod,
            check = "ncv")
```

```
## `geom_smooth()` using formula 'y ~ x'
```

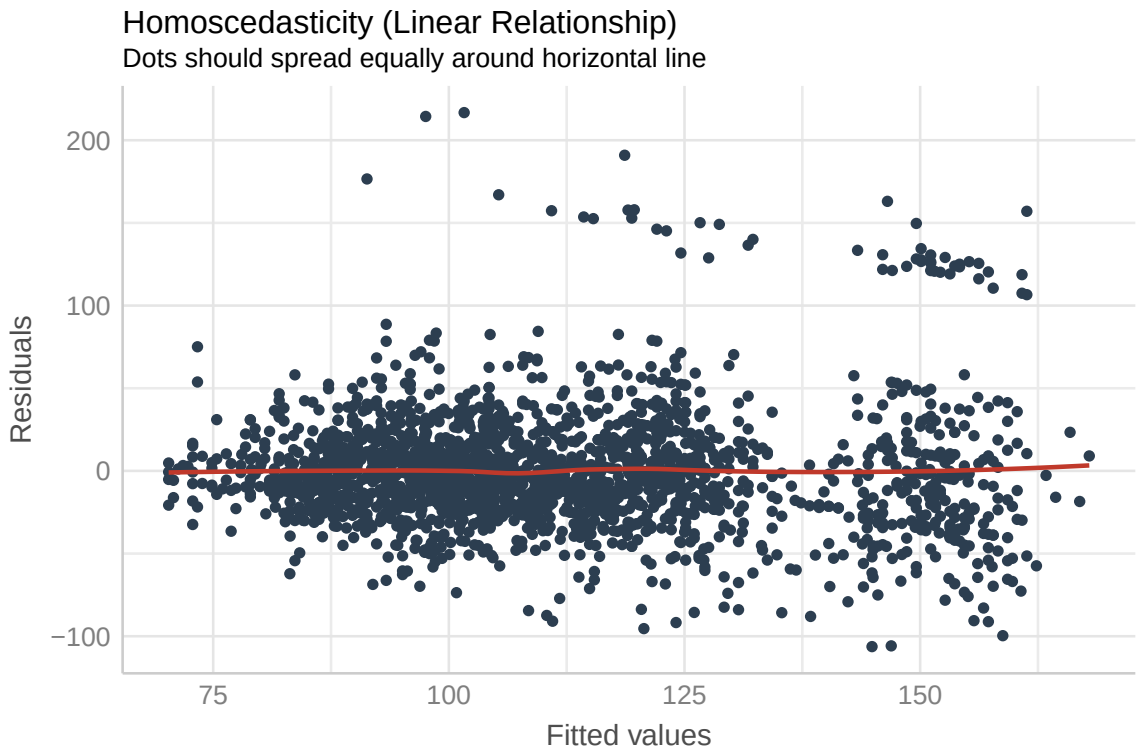

Figure 29: Homoscedasticity of the model with `performance`.

Those variables might be variables that we want to try to explore a little bit further and see if we can identify any other predictors in our data set that might be able to explain them.

9.3.1 Colour by variables not included in the model

For example, plotting by race, shows how some of the outliers may be explained by race.

```
ggplot(training, aes(x = finMod$fitted.values, y = finMod$residuals)) +
  geom_point(alpha = 0.5, aes(colour = race)) +
  geom_smooth(method = "loess", colour = "black", size = 0.5, se = FALSE) +
  labs(x = "Fitted values", y = "Residuals", colour = "Race") +
  theme_pubclean()
```

```
## `geom_smooth()` using formula 'y ~ x'
```

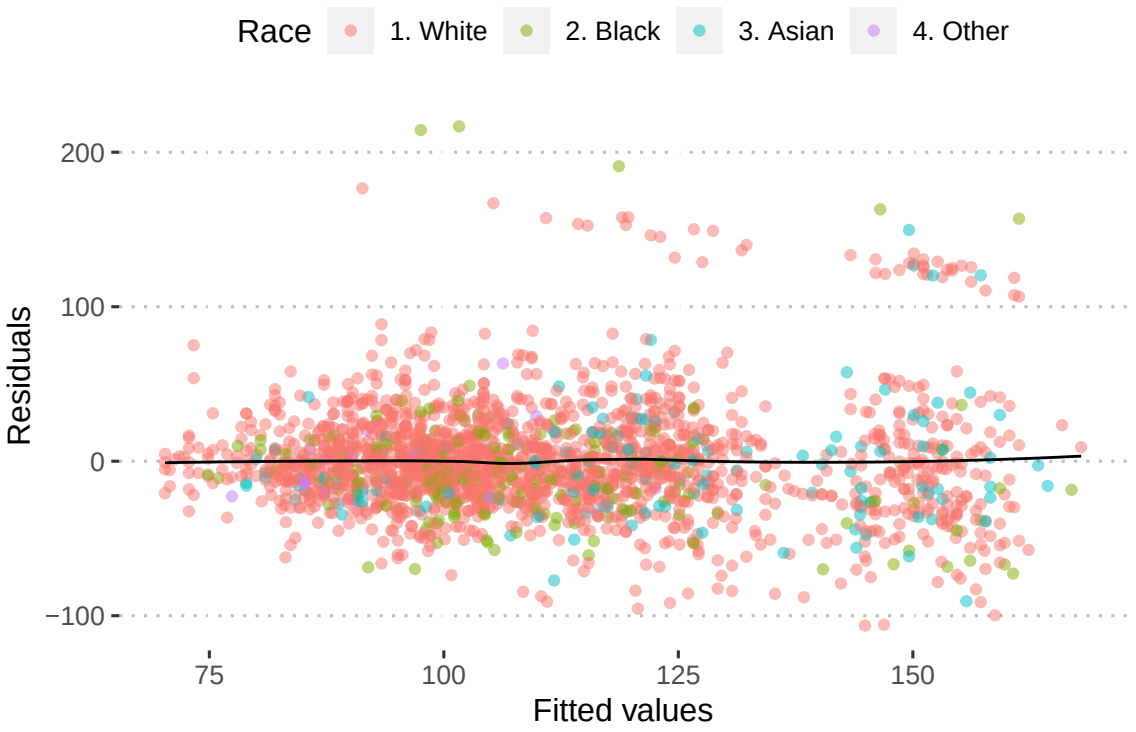

Figure 30: Association between wage and age by job race.

9.3.2 Plot by index

For example, plotting by race, shows how some of the outliers may be explained by race.

```
library(png)

img1 <- readPNG("indexresiduals.png")

im_A <- ggplot() + background_image(img1)
im_B <- ggplot(training, aes(x = c(1:length(finMod$residuals)),
                             y = finMod$residuals)) +
  geom_point(alpha = 0.5, aes(colour = race)) +
  geom_smooth(method = "loess", colour = "black", size = 0.5, se = FALSE) +
  labs(x = "Index", y = "Residuals", colour = "Race") +
  theme_pubclean()

ggarrange(im_A, im_B,
          ncol = 2,
          widths = c(1.06, 1),
          labels = "AUTO")
```

```
## `geom_smooth()` using formula 'y ~ x'
```

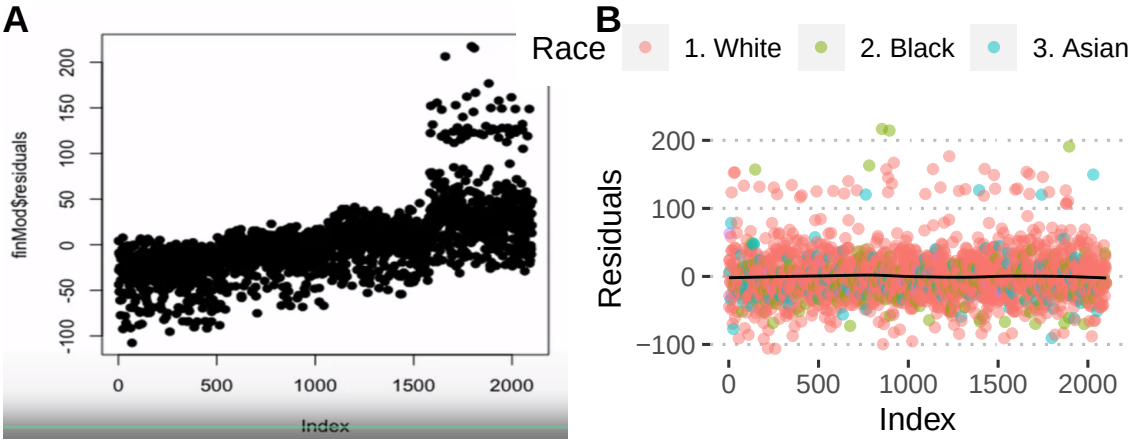

Figure 31: Index vs residuals. **A.** Plot produced in the lecture (I could not reproduce it). **B.** Plot created with `ggplot2`.

In panel **A**, high residuals seem to be happening at the right end of the highest row numbers.

There is a trend with respect to row numbers and so whenever you can see a trend or a outlier like that with respect to the row numbers, it suggests that there is a variable missing from your model because there should not be any relationship to the order in which the variables appear in the data set.

When you see a trend like this, or outliers like this, at one end of this plot, there's a relationship with respect to time, or age, or some other continuous variable that the rows are ordered by.

9.4 Predicted versus truth in the test set

To understand the model, plotting actual vs predicted values in the test set is a good option. Here, it is done by year, trying to understand where the outliers come from.

```
pred <- predict(modFit, testing)

ggplot(testing, aes(x = wage, y = pred, colour = year)) +
  geom_point(alpha = 0.5) +
  labs(x = "Wage (Actual)", y = "Wage (Predicted)", colour = "Year") +
  theme_pubclean() +
  theme(legend.position = "right")
```

Figure 32: Actual versus predicted Wage values in the test set.

Something to keep in mind is that if you do this sort of exploration in the test set, you cannot then go back and re-update your model in the training set because that would be using the test set to rebuild your predictors.

This is more like a post-mortem on your analysis or a way to try to determine whether your analysis worked or not.

9.5 If you want to use all covariates

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

```
## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading
```

```
## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading
```

```
## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading
```

```
## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading
```

```
pred <- predict(modFitAll, testing)
```

```
## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit
## may be misleading
```

```
ggplot(testing, aes(x = wage, y = pred)) +
  geom_point(alpha = 0.5) +
  labs(x = "Wage (Actual)", y = "Wage (Predicted)") +
  theme_pubclean()
```

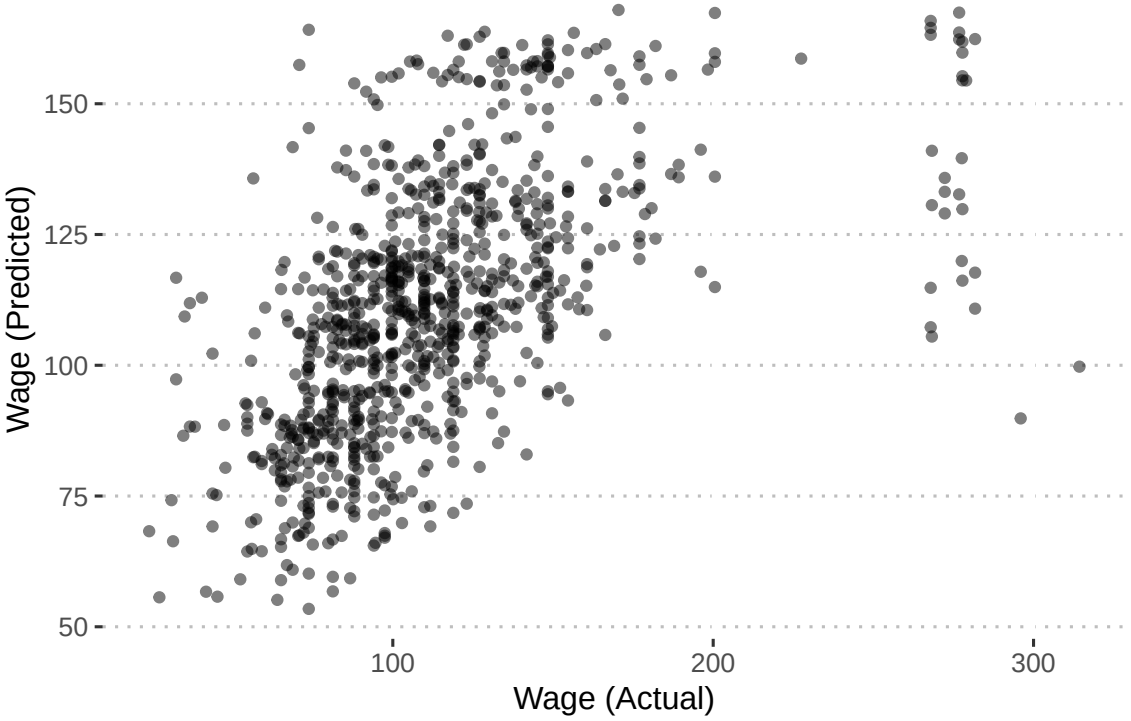

Figure 33: Actual versus predicted Wage values in the test set, from a model using all covariates.

9.6 Notes and further reading

- **Linear regression** is often useful in combination with other models
- **Exploratory analysis** (like the plots we made) are often very useful in helping identify predictors
- Elements of statistical learning (Hastie et al., 2009)
- Modern applied statistics with S (Venables & Ripley, 2002)
- Introduction to statistical learning (James et al., 2013)

References

Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). *The elements of statistical learning: Data mining, infer-*

ence, and prediction (2nd ed). Springer. https://web.stanford.edu/~hastie/ElemStatLearn//printings/ESLII_print12_toc.pdf

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning: With applications in R*. Springer. https://www.statlearning.com/s/ISLR-Seventh-Printing-xwa7.pdf

Kuhn, M. (n.d.). *A Short Introduction to the caret Package*. https://cran.r-project.org/web/packages/caret/vignettes/caret.html

Kuhn, M. (2013). Predictive Modeling with R and the caret Package. *The R User Conference 2013*, 63. https://www.r-project.org/conferences/useR-2013/Tutorials/kuhn/user_caret_2up.pdf

Kuhn, M. (2020a). *Caret package | R Documentation*. https://www.rdocumentation.org/packages/caret/versions/6.0-86

Kuhn, M. (2008). Building Predictive Models inRUsing thecaretPackage. *Journal of Statistical Software*, *28*(5). https://doi.org/10.18637/jss.v028.i05

Kuhn, M. (2019). *The `Caret` Package*. https://topepo.github.io/caret/

Kuhn, M. (2020b). *Caret: Classification and regression training*. https://CRAN.R-project.org/package=caret

Venables, W. N., & Ripley, B. D. (2002). *Modern applied statistics with s*. Springer New York. https://doi.org/10.1007/978-0-387-21706-2