CS109 Lab 1: Model fitting

Contents

R provides functions for fitting linear (`lm`) and generalized linear models (`glm`), local polynomial regression models (`loess`), and nonlinear least squares (`nls`) among others. Functions for fitting many other kinds of models are available in various R packages.

Formula

What most of these model fitting functions have in common is a way of specifying relationships among variables in the model in terms of a *formula*. A formula is usually defined using the tilde operator (`~`), with the response on the left and predictors on the right.

Thinking of the formula operator as a domain-specific language is helpful, because many things work differently inside a formula than they do in the rest of R. For example: - outside of a formula, `+` means "addition", but inside a formula `+` means "inclusion" - outside of a formula ":" is a sequence operator, but inside a formula it means "interaction" - outside a formula `(a + b + c)^2` means "square the sum of a, b, and c"; inside a formula it means "include a, b, c, and all two-way interactions between them"

Note that you can escape to the usual meaning with the `I` function: `I((a + b + c)^2)` means "square the sum of a, b, and c, even if inside a formula.

More details on the formula interface are described in the help page (`?formula`).

Model fitting examples

R comes with some built-in data sets that can be used for examples and demos. You can ask for the list of available data sets by calling `data()`. Here we will use the built-in *mtcars* data set.

```
data(mtcars)
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Fit a simple model predicting *miles per gallon* (`mpg`) from *horsepower* (`hp`) and *weight* (`wt`).

```
lm(mpg ~ hp + wt, data = mtcars)
```

```
##
## Call:
## lm(formula = mpg ~ hp + wt, data = mtcars)
##
## Coefficients:
## (Intercept)           hp           wt
##    37.22727     -0.03177     -3.87783
```

Add interaction between `hp` and `wt`.

```
lm(mpg ~ hp * wt, data = mtcars)
```

```
##
## Call:
## lm(formula = mpg ~ hp * wt, data = mtcars)
##
## Coefficients:
## (Intercept)           hp           wt        hp:wt
##    49.80842     -0.12010     -8.21662      0.02785
```

Add interactions between `hp` and *automatic transmission* (`am`), and between `wt` and `am`.

```
lm(mpg ~ (hp + wt) * am, data = mtcars)
```

```
##
## Call:
## lm(formula = mpg ~ (hp + wt) * am, data = mtcars)
##
## Coefficients:
## (Intercept)           hp           wt           am        hp:am
##    30.70393     -0.04094     -1.85591     13.74000      0.02779
##       wt:am
##    -5.76895
```

Your turn: predict mpg

1. Fit a linear model predicting *miles per gallon* (`mpg`) from number of cylinders (`cyl`)
2. Fit a linear model predicting *miles per gallon* (`mpg`) from number of cylinders (`cyl`), *displacement* (`disp`), *horsepower* (`hp`), and all two-way interactions among these three predictors. Do *not* include the three way interaction.
3. Number of cylinders and displacement are so highly correlated that it makes sense to think of them both as measures of *engine size*. Fit a model predicting mpg from the sum of `cyl` and `disp`.
4. (Advanced) Fit a generalized linear model predicting *automatic vs. manual transmission* (`am`) from *displacement* (`disp`).
5. (Advanced) Fit a generalized linear model predicting *automatic vs. manual transmission* (`am`) from *displacement* (`disp`) and *number of gears* (`gear`). Treat gear as an ordered factor and use polynomial contrasts.

Types, Classes and methods (lm, predict, resid)

We've seen how to use *formula* in R to fit linear models, but you may have been somewhat underwhelmed by the result. `lm` returns simply the *Call* that produced it, and the regression coefficients. To do anything useful with these models we will want to assign them to a name and then do some post-estimation calculations. In order to do that we need to know what `lm` returns, and what methods exist for it. Most of this is documented in `?lm`, but we can use R to inspect the result and learn for ourselves.

Objects produced by `lm` are named *lists*, and can be treated just like any other list in R. Let's simplify our last example model and take a closer look.

```
mod.mpg1 <- lm(mpg ~ wt * am, data = mtcars)
typeof(mod.mpg1)
```

```
## [1] "list"
```

```
length(mod.mpg1)
```

```
## [1] 12
```

```
names(mod.mpg1)
```

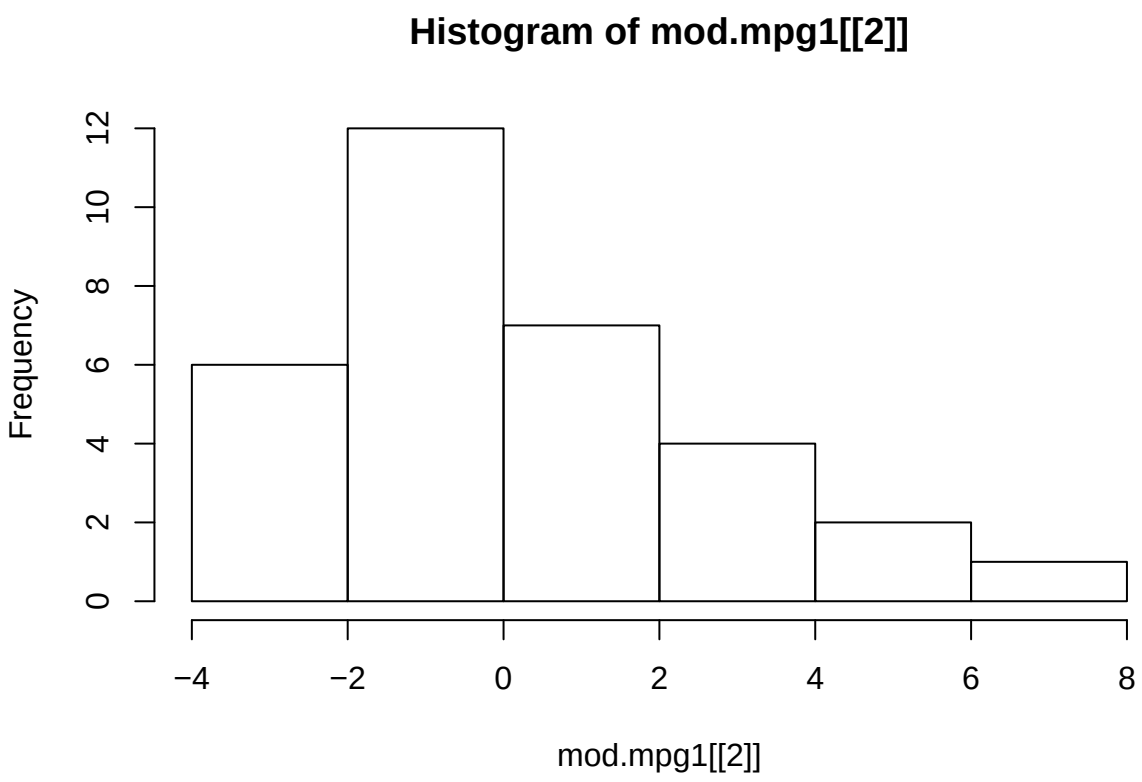
```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

Since `mod.mpg1` is a list, we can do anything with it that we can do with other lists in R. For example, we can extract elements by position or name:

```
mod.mpg1[["coefficients"]]
```

```
## (Intercept)          wt          am       wt:am
##   31.416055   -3.785908   14.878423   -5.298360
```

```
hist(mod.mpg1[[2]])
```



Unlike other lists in R, lists produced by `lm` are of *class* `lm`. There are functions with methods that correspond to this class, and we can ask R to tell us what those functions are using the `methods` function.

```
class(mod.mpg1)
```

```
## [1] "lm"
```

```
methods(class = class(mod.mpg1))
```

```
##  [1] add1           alias          anova          case.names
##  [5] coerce         confint        cooks.distance deviance
##  [9] dfbeta         dfbetas        drop1          dummy.coef
## [13] effects        extractAIC     family         formula
## [17] hatvalues      influence      initialize     kappa
## [21] labels         logLik         model.frame    model.matrix
## [25] nobs           plot           predict        print
## [29] proj           qr             residuals      rstandard
## [33] rstudent       show           simulate       slotsFromS3
## [37] summary        variable.names vcov
## see '?methods' for accessing help and source code
```

We now know that there are `summary`, `anova`, and `confint` methods (among others) for objects of class `lm`. That is, we know that we can do the following:

```
summary(mod.mpg1)
```

```
##
## Call:
## lm(formula = mpg ~ wt * am, data = mtcars)
```

```
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.6004 -1.5446 -0.5325  0.9012  6.0909
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  31.4161     3.0201  10.402 4.00e-11 ***
## wt           -3.7859     0.7856  -4.819 4.55e-05 ***
## am           14.8784     4.2640   3.489  0.00162 **
## wt:am        -5.2984     1.4447  -3.667  0.00102 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.591 on 28 degrees of freedom
## Multiple R-squared:  0.833,  Adjusted R-squared:  0.8151
## F-statistic: 46.57 on 3 and 28 DF,  p-value: 5.209e-11
```

```
anova(mod.mpg1)
```

```
## Analysis of Variance Table
##
## Response: mpg
##           Df Sum Sq Mean Sq  F value    Pr(>F)
## wt         1 847.73  847.73 126.2518 6.915e-12 ***
## am         1   0.00    0.00   0.0003  0.985556
## wt:am      1  90.31   90.31  13.4502  0.001017 **
## Residuals 28 188.01    6.71
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
confint(mod.mpg1)
```

```
##                 2.5 %    97.5 %
## (Intercept) 25.229642 37.602469
## wt          -5.395234 -2.176581
## am           6.143928 23.612917
## wt:am       -8.257693 -2.339028
```

Note In Python the equivalent of `summary(mod.mpg1)` would probably look like `mod.mpg1.summary()`. This *object-oriented* style does exist in R (though it uses `$` instead of `.` to access methods), but it is not commonly used.

This technique of identifying the *class* of an R object, and then looking up which generic functions have corresponding *methods* is very useful. Remember thought that it only shows you functions that have specific methods; for example `methods(class = class(mod.mpg1))` did not show us a method for `[[`, but we know that we can use bracket extraction on a *lm* object. That just means that `[[` doesn't do anything special because of the object's class – it just treats it like any other list.

We can also go the other way around, and ask R what methods exist for a specific generic function. For example,

```
methods(summary)
```

```
##  [1] summary.aov                   summary.aovlist*
##  [3] summary.aspell*               summary.check_packages_in_dir*
##  [5] summary.connection            summary.data.frame
##  [7] summary.Date                  summary.default
```

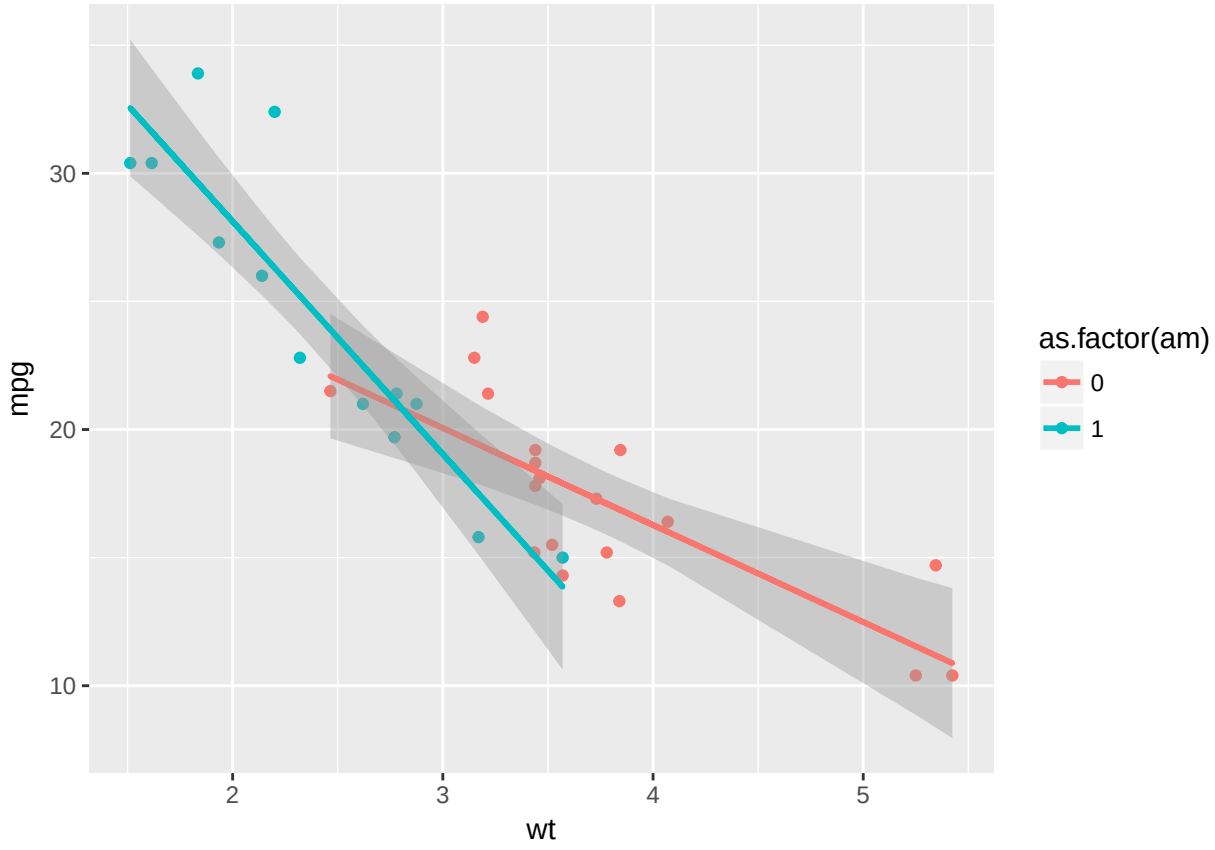
```
##  [9] summary.ecdf*                  summary.factor
## [11] summary.glm                    summary.infl*
## [13] summary.lm                     summary.loess*
## [15] summary.manova                 summary.matrix
## [17] summary.mlm*                   summary.nls*
## [19] summary.packageStatus*         summary.PDF_Dictionary*
## [21] summary.PDF_Stream*            summary.POSIXct
## [23] summary.POSIXlt                summary.ppr*
## [25] summary.prcomp*                summary.princomp*
## [27] summary.proc_time              summary.srcfile
## [29] summary.srcref                 summary.stepfun
## [31] summary.stl*                   summary.table
## [33] summary.tukeysmooth*
## see '?methods' for accessing help and source code
```

shows us all the different object classes that have `summary` methods.

Most modeling functions in R usually return objects with at least `summary`, `print`, and `predict` methods. The `predict` methods are useful for visualizing the quality of your model. Once you've calculated predicted values you can use the `plot` or `ggplot` functions to construct a graph. Let's use `ggplot` to visualize the predictions from our last model.

```
library(ggplot2)

ggplot(transform(mtcars, pred.mpg = predict(mod.mpg1, interval = "confidence")),
       mapping = aes(x = wt, y = mpg, color = as.factor(am))) +
    geom_point() +
    geom_smooth(mapping = aes(y = pred.mpg.fit,
                              ymin = pred.mpg.lwr,
                              ymax = pred.mpg.upr),
                stat = "identity")
```

```
## Warning: Ignoring unknown aesthetics: ymin, ymax
```



Your turn: Get to know the *lm* class

1. Fit a model predicting `mpg` from `disp`, assigning the result to the name `mod.mpg`.
2. Get the `summary` of the model and assign the result to the name `mod.mpg.sum`.
3. Find the `typeof` and `names` of `mod.mpg.sum`.
4. Extract the `r.squared` value from `mod.mpg.sum`.
5. Graph the observed and predicted values of `mpg` as a function of `disp` based on your model.
6. (Advanced) Inference from linear regression models is based on several assumptions, including the assumption that the residuals are normally distributed. Visually inspect your model to determine if this assumption has been violated.
7. (Advanced) Add `am` and the interaction between `disp` and `am` to your model. What is the regression coefficient for `mpg` on `disp` when `am = 0`? When `am = 1`?

Transformations (Polynomials and Splines)

The R *formula* provides a way to specify relationships among variables, and among *transformations* of those variables. We've already seen that `*` transforms the inputs by including both the specified variables and their product. Many other transformations are possible. Indeed, for categorical predictors transformation is mandatory and happens by default; R will transform categorical predictors into a matrix of indicators or dummy codes with the first level as the reference group.

In general, any transformation that returns a vector of length equal to the number of rows in the data or a matrix with the same row dimension as the input data is legal.

Standardized regression coefficients

A example of transformation is the `scale` function, which standardizes it's first argument by subtracting the mean and dividing by the standard deviation:

```
str(mtcars[c("hp", "wt")])
```

```
## 'data.frame':    32 obs. of  2 variables:
##  $ hp: num  110 110 93 110 175 105 245 62 95 123 ...
##  $ wt: num  2.62 2.88 2.32 3.21 3.44 ...
```

```
str(scale(mtcars[c("hp", "wt")]))
```

```
##  num [1:32, 1:2] -0.535 -0.535 -0.783 -0.535 0.413 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
##   ..$ : chr [1:2] "hp" "wt"
##  - attr(*, "scaled:center")= Named num [1:2] 146.69 3.22
##   ..- attr(*, "names")= chr [1:2] "hp" "wt"
##  - attr(*, "scaled:scale")= Named num [1:2] 68.563 0.978
##   ..- attr(*, "names")= chr [1:2] "hp" "wt"
```

Thus if we want standardized regression coefficients we can fit our model as follows:

```
mod.mpg2 <- lm(mpg ~ scale(cbind(hp = hp, wt = wt)) * am, data = mtcars)
```

Polynomials

If we want polynomial terms we can generate them ourselves, or use the `poly` function to do it for us.

```
summary(lm(mpg ~ disp + I(disp^2) + I(disp^3), data = mtcars))
```

```
##
## Call:
## lm(formula = mpg ~ disp + I(disp^2) + I(disp^3), data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0896 -1.5653 -0.3619  1.4368  4.7617
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  5.070e+01  3.809e+00  13.310 1.25e-13 ***
## disp        -3.372e-01  5.526e-02  -6.102 1.39e-06 ***
## I(disp^2)    1.109e-03  2.265e-04   4.897 3.68e-05 ***
## I(disp^3)   -1.217e-06  2.776e-07  -4.382  0.00015 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.224 on 28 degrees of freedom
## Multiple R-squared:  0.8771, Adjusted R-squared:  0.8639
## F-statistic: 66.58 on 3 and 28 DF,  p-value: 7.347e-13
```

```
(mod.mpg3 <- lm(mpg ~ poly(disp, degree = 3, raw = TRUE), data = mtcars))
```

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

```
## lm(formula = mpg ~ poly(disp, degree = 3, raw = TRUE), data = mtcars)
##
## Coefficients:
##                          (Intercept)  poly(disp, degree = 3, raw = TRUE)1
##                            5.070e+01                             -3.372e-01
## poly(disp, degree = 3, raw = TRUE)2  poly(disp, degree = 3, raw = TRUE)3
##                            1.109e-03                             -1.217e-06
```

```
summary(mod.mpg3)
```

```
##
## Call:
## lm(formula = mpg ~ poly(disp, degree = 3, raw = TRUE), data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0896 -1.5653 -0.3619  1.4368  4.7617
##
## Coefficients:
##                                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)                          5.070e+01  3.809e+00  13.310 1.25e-13
## poly(disp, degree = 3, raw = TRUE)1 -3.372e-01  5.526e-02  -6.102 1.39e-06
## poly(disp, degree = 3, raw = TRUE)2  1.109e-03  2.265e-04   4.897 3.68e-05
## poly(disp, degree = 3, raw = TRUE)3 -1.217e-06  2.776e-07  -4.382  0.00015
##
## (Intercept)                         ***
## poly(disp, degree = 3, raw = TRUE)1 ***
## poly(disp, degree = 3, raw = TRUE)2 ***
## poly(disp, degree = 3, raw = TRUE)3 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.224 on 28 degrees of freedom
## Multiple R-squared:  0.8771, Adjusted R-squared:  0.8639
## F-statistic: 66.58 on 3 and 28 DF,  p-value: 7.347e-13
```

Splines

Similarly, we can use the `bs` function from the `splines` package to generate piece-wise polynomials. You can specify either `df` or you can specify `knots` directly (in which case `ns` will choose `df - 1 - intercept` knots at suitable intervals).

```
library(splines)
mod.mpg4 <- lm(mpg ~ bs(disp, knots = quantile(disp, c(.25, .50, .75))), data = mtcars)
summary(mod.mpg4)
```

```
##
## Call:
## lm(formula = mpg ~ bs(disp, knots = quantile(disp, c(0.25, 0.5,
##     0.75))), data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4422 -1.4306 -0.2142  1.4879  3.8917
##
```

```
## Coefficients:
##                                                      Estimate Std. Error
## (Intercept)                                            32.420      1.601
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))1   -3.488      3.593
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))2  -13.042      2.267
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))3  -10.341      3.781
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))4  -22.558      3.559
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))5  -10.374      3.907
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))6  -23.206      2.328
##                                                      t value Pr(>|t|)
## (Intercept)                                           20.252  < 2e-16 ***
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))1  -0.971   0.3409
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))2  -5.754 5.38e-06 ***
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))3  -2.735   0.0113 *
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))4  -6.338 1.24e-06 ***
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))5  -2.655   0.0136 *
## bs(disp, knots = quantile(disp, c(0.25, 0.5, 0.75)))6  -9.969 3.41e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.136 on 25 degrees of freedom
## Multiple R-squared:  0.8987, Adjusted R-squared:  0.8744
## F-statistic: 36.97 on 6 and 25 DF,  p-value: 2.986e-11
```

Again there is nothing magical about splines, they are basically piece-wise polynomials. We can calculate a quick-and-dirty version ourselves without too much trouble:

```
disp.qtl <- quantile(mtcars$disp, c(.25, .50, .75))
all.equal(predict(lm(mpg ~ disp + I(disp^2) + I(disp^3) +
                         ifelse(disp > disp.qtl[1], (disp - disp.qtl[1])^3, 0) +
                         ifelse(disp > disp.qtl[2], (disp - disp.qtl[2])^3, 0) +
                         ifelse(disp > disp.qtl[3], (disp - disp.qtl[3])^3, 0),
                     data = mtcars)),
          predict(mod.mpg4))
```

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

Finally, we can construct natural cubic splines using the `ns` function. As with `bs` you can specify either `df` or you can specify `knots` directly.

What all of these transformation functions (i.e., `scale`, `poly`, `bs`, `ns`) have in common is that they take a variable as input and return a matrix where the row dimension is equal to the length of the input variable. When called inside a `formula` of a modeling function (e.g., `lm`) the generated matrix will be used in the right-hand-side of the equation.

Your turn: Visualizing polynomial and spline regressions

1. Predict `mpg` from `hp` using a *natural spline* (`ns`) with `df = 3`.
2. Predict `mpg` from `hp` using a *natural spline* with `df = 8`.
3. Plot the predictions from these two models.
4. Extract the R-square values from each model.
5. (Advanced) Refit the `df=3` model, adding an interaction with `am`. Plot the predictions, coloring by `am`.

Selecting parameter values

Once we start fitting models with polynomials or splines, we will quickly realize that we need a way to choose the optimal parameters. What degree polynomial should we use? How many degrees of freedom in our basis-splines?

A simple answer is to choose a measure of model fit, test out a range of parameters, and see which one fits the best. That makes sense, but be careful not to over-fit the model. In other words, you want you model to capture generalizable patterns, and *not* to capture noise in the sample you happen to have.

Basic cross-validation

There are several approaches to avoiding over-fitting, with cross-validation being a simple and common method. Here are the basic steps.

1. Split the data into training and test sets.
2. Choose a measure of model fit, e.g., R^2 or $rmse$.
3. Fit each model to the training set.
4. Calculate the measure of model fit on the *test* set.

Here is an example using the `mtcars` data.

```
## define function to calculate R square
rsq <- function(model, data, y) {
    1 - (var(residuals(model, data, type = "response")) / var(data[[y]], na.rm = TRUE))
}
```

Split the data into test and training sets.

```
id.train <- sample(1:nrow(mtcars), floor(nrow(mtcars) * 0.75))
mt.train <- mtcars[id.train, ]
mt.test <- mtcars[-id.train, ]
```

Next, define a function to fit a model with given df and calculate R-square.

```
model.performance <- function(df, train, test) {
    mod <- lm(mpg ~ bs(disp, df = df), data = train)
    c(train.r2 = rsq(mod, train, "mpg"),
      test.r2 = rsq(mod, test, "mpg"))
}
```

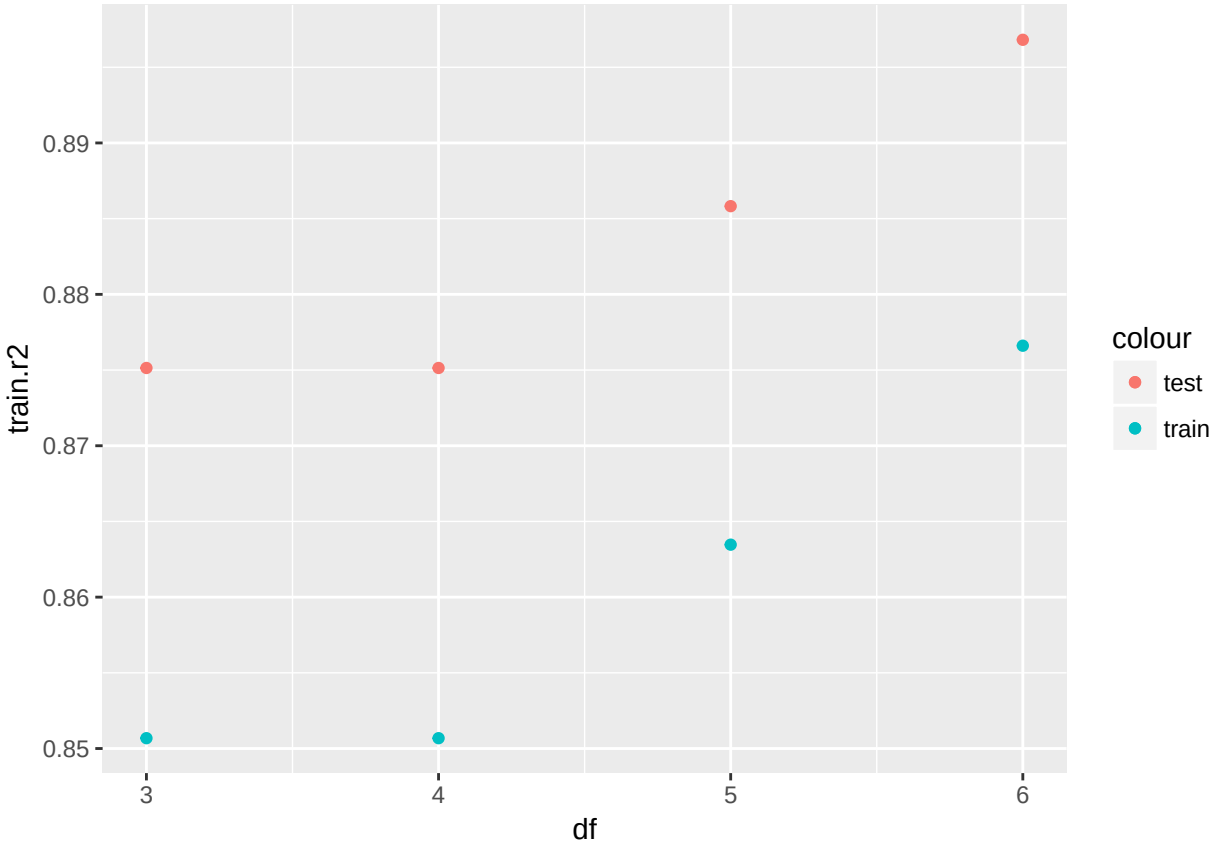
Finally, fit the model with varying parameters and calculate performance for each model

```
dfs <- 3:6 ## parameters we want to iterate over

## fit models using sapply. We could have used a for-loop instead.
performance <- sapply(dfs, model.performance,
                      train = mt.train,
                      test = mt.test,
                      simplify = FALSE)

## arrange the result in a data.frame
performance <- as.data.frame(do.call(rbind, performance))
performance$df <- dfs

performance
```

```
##    train.r2   test.r2 df
## 1 0.8506826 0.8751361  3
## 2 0.8506829 0.8751363  4
## 3 0.8634651 0.8858252  5
## 4 0.8766071 0.8968150  6
```

Usually it's a good idea to visualize model performance as a function of the varying parameter(s).

```
library(ggplot2)

ggplot(performance, aes(x = df)) +
    geom_point(aes(y = train.r2, color = "train")) +
    geom_point(aes(y = test.r2, color = "test"))
```



k-fold cross-validation

There is a serious limitation to using a single test/training split to perform cross validation. Would you get the same result with a different split? Because we don't want our results to depend on the random partitioning into test and training sets, it is a good idea to use multiple splits. This leads us to k-fold cross-validation. The basic steps are:

1. Split the data into `k` paritions.
2. Choose a measure of model fit, e.g., R^2 or $rmse$.
3. For each partition, fit the model to the data excluding that partition.
4. Calculate the measure of model fit on the excluded partition.
5. Average the `k` measures of model fit.

Here is an example, again using the `mtcars` data.

Create `k=5` partitions.

```
k <- 5

mtcars$partition <- cut(sample(1:nrow(mtcars), nrow(mtcars)), 5)
```

Fit the model with varying parameters and calculate performance for each model.

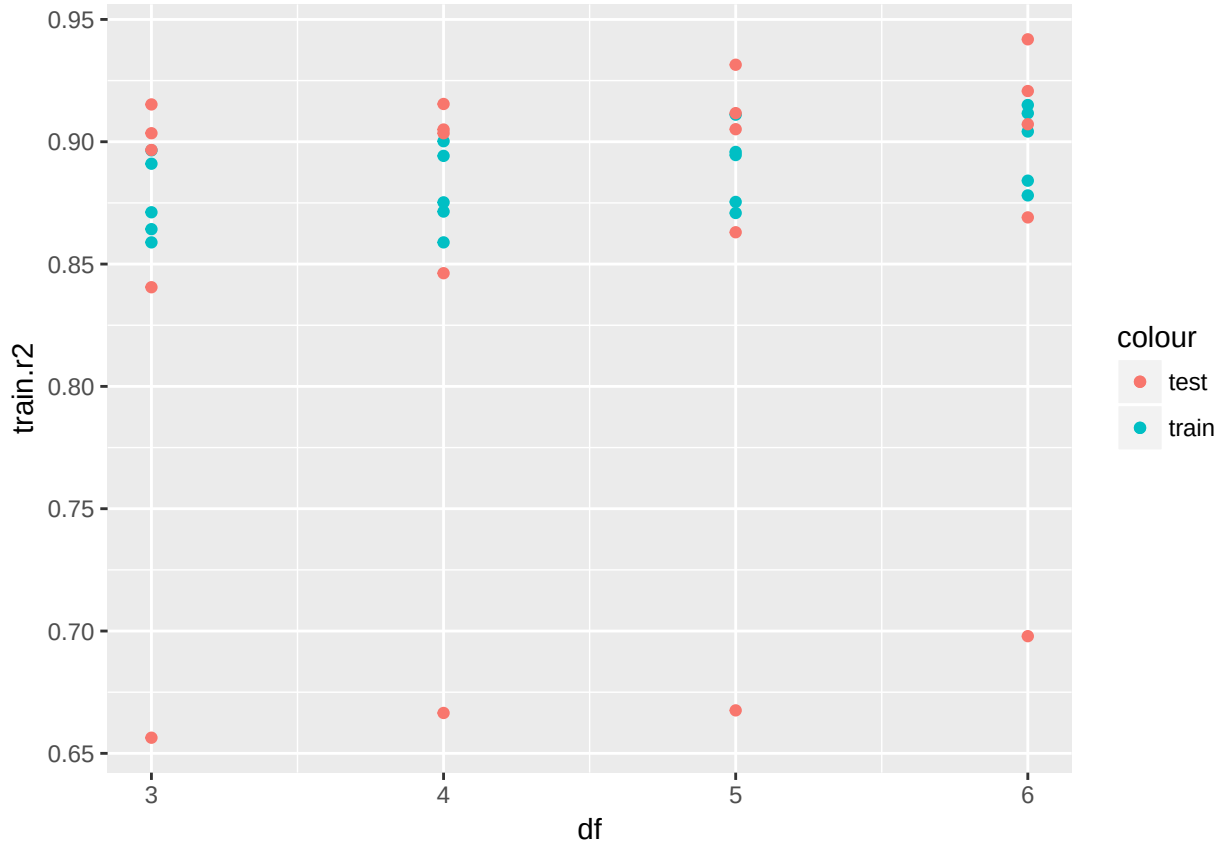
```
dfs <- 3:6 ## parameters we want to iterate over

## Use a for-loop to iterate over the partitions.
## We could have used sapply instead
performance <- vector(mode = "list", length = k)
names(performance) <- unique(mtcars$partition)
for(partition in names(performance)) {
    ## Fit models using sapply. We could have used a for-loop instead.
    test <- mtcars$partition == partition
    performance[[partition]] <- sapply(dfs,
                                       model.performance,
                                       train = mtcars[!test, ],
                                       test = mtcars[test, ],
                                       simplify = FALSE)
}

## arrange the result in a list of data.frames
performance <- sapply(performance, function(x) {
    x <- as.data.frame(do.call(rbind, x))
    x$df <- dfs
    x},
    simplify = FALSE)

## add partition column
for(partition in names(performance)) {
    performance[[partition]] <- data.frame(performance[[partition]],
                                           partition = partition)
}

## reduce list of data.frames to a single data.frame
performance <- do.call(rbind, performance)

performance
```

```
##                train.r2   test.r2 df   partition
## (25.8,32].1   0.8588712 0.9034839  3   (25.8,32]
## (25.8,32].2   0.8588712 0.9034839  4   (25.8,32]
## (25.8,32].3   0.8708804 0.9116968  5   (25.8,32]
## (25.8,32].4   0.8840922 0.9207322  6   (25.8,32]
## (19.6,25.8].1 0.8964960 0.8405342  3 (19.6,25.8]
## (19.6,25.8].2 0.9002199 0.8462717  4 (19.6,25.8]
## (19.6,25.8].3 0.9110832 0.8630083  5 (19.6,25.8]
## (19.6,25.8].4 0.9150266 0.8690839  6 (19.6,25.8]
## (13.4,19.6].1 0.8910234 0.6563962  3 (13.4,19.6]
## (13.4,19.6].2 0.8942303 0.6665077  4 (13.4,19.6]
## (13.4,19.6].3 0.8945673 0.6675701  5 (13.4,19.6]
```

```
## (13.4,19.6].4 0.9041869 0.6979009  6 (13.4,19.6]
## (7.2,13.4].1  0.8642654 0.8966533  3  (7.2,13.4]
## (7.2,13.4].2  0.8752273 0.9049996  4  (7.2,13.4]
## (7.2,13.4].3  0.8753990 0.9051303  5  (7.2,13.4]
## (7.2,13.4].4  0.8780709 0.9071647  6  (7.2,13.4]
## (0.969,7.2].1 0.8711783 0.9152512  3 (0.969,7.2]
## (0.969,7.2].2 0.8714659 0.9154404  4 (0.969,7.2]
## (0.969,7.2].3 0.8958383 0.9314744  5 (0.969,7.2]
## (0.969,7.2].4 0.9116469 0.9418746  6 (0.969,7.2]
```

Usually it's a good idea to visualize model performance as a function of the varying parameter(s).

```
library(ggplot2)

ggplot(performance, aes(x = df)) +
    geom_point(aes(y = train.r2, color = "train")) +
    geom_point(aes(y = test.r2, color = "test"))
```



Note: the `caret` package is a full-featured system for training regression and classification models. The `modelr` package is a light-weight alternative. In this exercise we are doing the cross-validation ourselves in order to better understand how it works.

Your turn: Use cross validation to determine the optimal degrees of freedom

Note You may use the convenience functions we wrote earlier, or write your own.

1. Split the `mtcars` data into a test and training set.
2. Iterate over `df` from 3-6, predicting `mpg` from `hp` using a *natural spline* (`ns`). For each `df` calculate the R^2 value in the test and training set.
3. Plot the R^2 values in the test and training set as a function of `df`. What is the best `df` setting?
4. (Advanced) Repeat steps 1-3 several times. Do you get the same answer with different train/test splits?
5. (Advanced) Use 5-fold cross-validation to select the optimal df.

Smoothing models (loess, smooth.spline, gam)

As flexible as the *transformation* mechanism described above is, it is limited. Other approaches to modeling non-linear functions include local polynomial regression (`loess`) and generalized additive models (`gam`). Rather than using standard (generalized) linear models with transformations on the right-hand-side, these models are best fit in R using specialized functions.

Local polynomial regression works by fitting a polynomial weighted by distance from the point being predicted. The most important parameters are `span` and `degree`, which control the down-weighting of more distant points, and the degree of the polynomial, respectively.

```
mod.mpg5 <- loess(mpg ~ disp, data = mtcars) # default span = 0.75
mod.mpg6 <- loess(mpg ~ disp, span = 0.25, data = mtcars) # less smooth
mod.mpg7 <- loess(mpg ~ disp, span = 2.0, data = mtcars) # more smooth
```

The `smooth.spline` function is a bit of an oddball in that it does not have a formula interface. Instead we specify `x` and `y` arguments, along with a smoothing parameter `spar`.

There are two popular implementations of generalized additive models in R, the older *gam* package, and the more recent *mgcv* package. *mgcv*. We'll use *gam* because it is somewhat simpler and easier to understand.

Calls go the `gam` function look much like calls to `lm` with transformations that we saw earlier:

```
library(gam)
```

```
## Loading required package: foreach
```

```
## Loaded gam 1.14
```

```
mod.mpg8 <- gam(mpg ~ s(disp, spar = 0.1), data = mtcars)
```

Under the hood things are a bit different. The `s` function doesn't actually compute the smooth splines – it just adds some attributes so that the `gam` function knows how to generate the model matrix. In other words the `s` function is *specific* to the *gam* package: `lm(mpg ~ s(disp, spar = 0.1), data = mtcars)` will not work.

Your turn: Use cross validation to determine the optimal degrees of freedom

Note You may modify and use the convenience functions we wrote earlier, or write your own.

1. Split the `mtcars` data into a test and training set.

2. Iterate over `spar` from 0.1 – 0.5 (in increments of 0.1), predicting `mpg` from `hp` using a *generalized additive model* (`gam`). For each value of`spar` calculate the R^2 value in the test and training set.
3. Plot the R^2 values in the test and training set as a function of `spar`. What is the best `spar` setting?
4. (Advanced) Repeat steps 1-3 several times. Do you get the same answer with different train/test splits?
5. (Advanced) Use 5-fold cross-validation to select the optimal `spar` value.