Advanced Modeling

Kylie Ariel Bemis

2/25/2019

Advanced Modeling

> *All models are wrong, but some are useful.*

– George Box

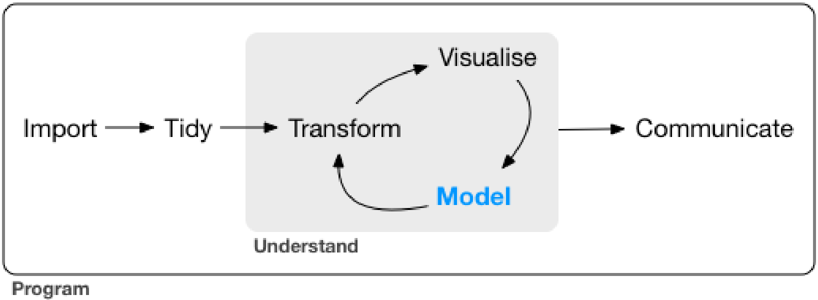


Figure 1: Wickham and Grolemund, *R for Data Science*

Suppose we want to predict flight arrival delays

```r
library(nycflights13)
flights2 <- transmute(flights,
                      month = factor(month,
                                     levels=1:12),
                      dep_time = factor(dep_time %/% 100,
                                        levels=0:23),
                      arr_delay, dep_delay,
                      origin, dest, distance)
library(modelr)
set.seed(1)  # remember to set seeds for reproducibility!
flights3 <- resample(flights2, sample(nrow(flights), 20000))
flights3 <- mutate(as_tibble(flights3), rand=rnorm(20000))
flights_part <- resample_partition(flights3, c(train = 0.8,
                                               valid = 0.1,
                                               test = 0.1))
flights4 <- as_tibble(flights_part$train)
```

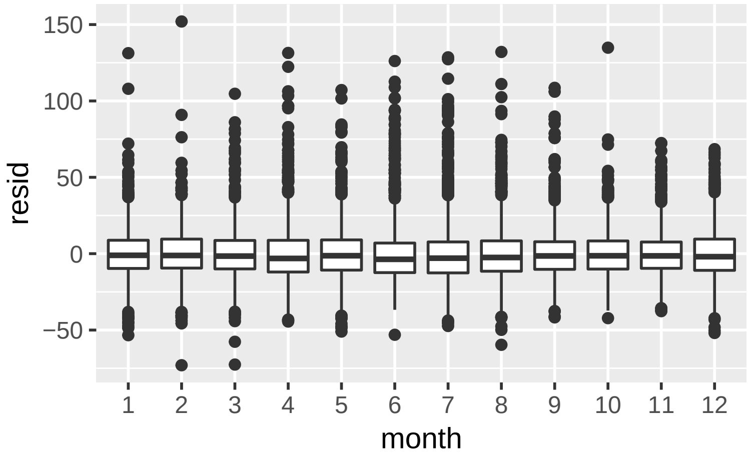
Fit a model

```
fit1 <- lm(arr_delay ~ month + dep_time + dep_delay + distance,
           data=flights4)
fit1
```

```
##
## Call:
## lm(formula = arr_delay ~ month + dep_time + dep_delay + distance,
##     data = flights4)
##
## Coefficients:
## (Intercept)       month2       month3       month4       month5
##   -7.972954    -1.425768    -2.838335     1.435123    -5.191636
##      month6       month7       month8       month9      month10
##    1.004984    -0.562892    -2.166221    -6.327614    -1.701009
##     month11      month12    dep_time1    dep_time2    dep_time3
##   -0.238005     1.966489     2.780057    -6.188527   -21.438390
##   dep_time4    dep_time5    dep_time6    dep_time7    dep_time8
##    8.365532     6.518789     6.087995     4.025898     6.564085
##   dep_time9   dep_time10   dep_time11   dep_time12   dep_time13
##    6.304816     5.152280     5.616925     6.262925     6.613683
##  dep_time14   dep_time15   dep_time16   dep_time17   dep_time18
##    5.889116     8.168880     5.129967     5.098119     6.397573
```

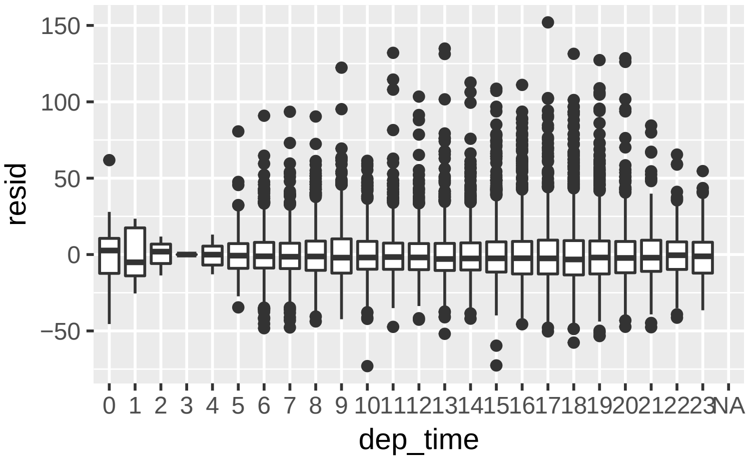
Residuals vs month

```
flights4 %>% add_residuals(fit1) %>%
  ggplot(aes(x=month, y=resid)) + geom_boxplot()
```



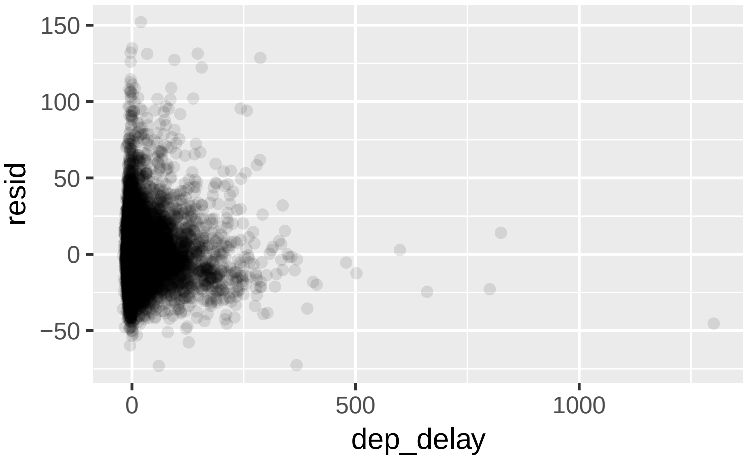
Residuals vs dep_time

```
flights4 %>% add_residuals(fit1) %>%
  ggplot(aes(x=dep_time, y=resid)) + geom_boxplot()
```



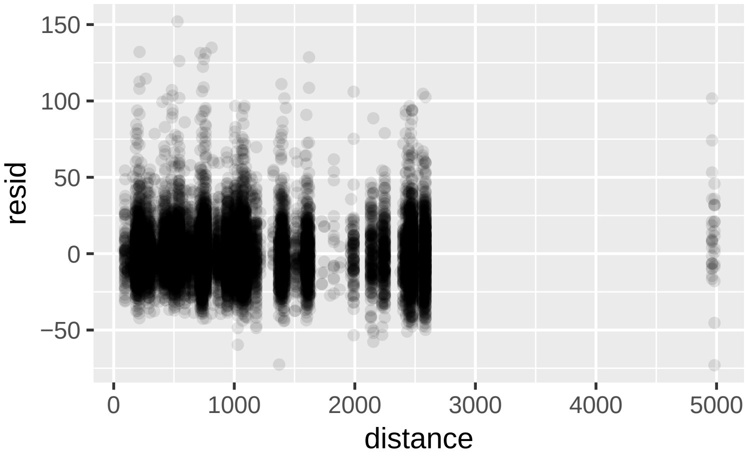
Residuals vs dep_delay

```
flights4 %>% add_residuals(fit1) %>%
  ggplot(aes(x=dep_delay, y=resid)) + geom_point(alpha=0.1)
```



Residuals vs distance

```
flights4 %>% add_residuals(fit1) %>%
  ggplot(aes(x=distance, y=resid)) + geom_point(alpha=0.1)
```



Check predictive error on test set

```r
rmse(fit1, flights_part$train)
```

```
## [1] 17.77036
```

```r
rmse(fit1, flights_part$test) # typically higher on test set
```

```
## [1] 17.38187
```

```r
mae(fit1, flights_part$train)
```

```
## [1] 12.891
```

```r
mae(fit1, flights_part$test) # typically higher on test set
```

```
## [1] 12.81089
```

Cross-validation

What if we want to use all of the data for both training and testing?

Cross-validation accomplishes this while avoiding over-fitting.

To perform k-fold cross-validation:

- Partition the data into k subsets
- Repeat k times:
  - Hold one of the k subsets for testing
  - Train model on the other pool of k-1 subsets
  - Test the model on the subset held for testing
- Calculate the average performance over all k folds

Visualizing cross-validation

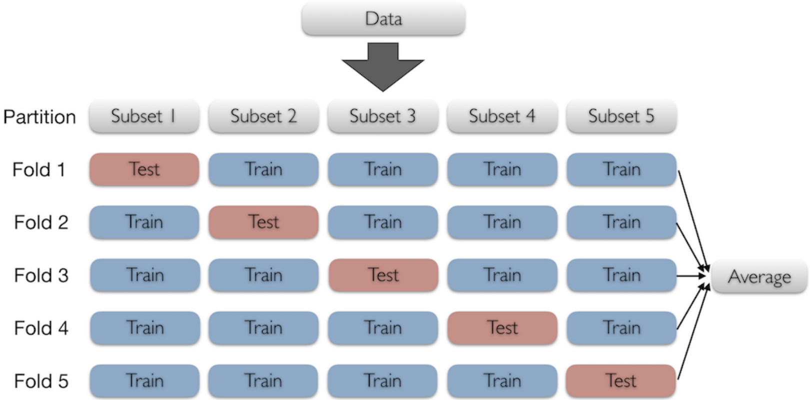


Figure 2: 5-fold cross-validation

Partition the data for cross-validation

```r
set.seed(1) # remember to set seeds for reproducibility!
flights_cv <- crossv_kfold(flights2, 10)
flights_cv
```

```
## # A tibble: 10 x 3
##    train          test           .id
##    <list>         <list>         <chr>
##  1 <S3: resample> <S3: resample> 01
##  2 <S3: resample> <S3: resample> 02
##  3 <S3: resample> <S3: resample> 03
##  4 <S3: resample> <S3: resample> 04
##  5 <S3: resample> <S3: resample> 05
##  6 <S3: resample> <S3: resample> 06
##  7 <S3: resample> <S3: resample> 07
##  8 <S3: resample> <S3: resample> 08
##  9 <S3: resample> <S3: resample> 09
## 10 <S3: resample> <S3: resample> 10
```

What are these `train` and `test` columns?

List-columns

It is perfectly valid to use `lists` as a column in a data.frame.

R does not usually make this easy, but `tibble` simplifies the process.

```
data.frame(a=list(1:3, 4:6, 7:9), b=c("a", "b", "c"))
```

```
##   a.1.3 a.4.6 a.7.9 b
## 1     1     4     7 a
## 2     2     5     8 b
## 3     3     6     9 c
```

```
tibble(a=list(1:3, 4:6, 7:9), b=c("a", "b", "c"))
```

```
## # A tibble: 3 x 2
##   a         b
##   <list>    <chr>
## 1 <int [3]> a
## 2 <int [3]> b
## 3 <int [3]> c
```

How do we operate on lists?

```r
l <- list(a=1:4, b=5:8, c=9:12)
l
```

```
## $a
## [1] 1 2 3 4
##
## $b
## [1] 5 6 7 8
##
## $c
## [1]  9 10 11 12
```

Applying functions to lists

R gives us a few native ways to do it:

```r
lapply(l, mean) # "list-apply"
```

```
## $a
## [1] 2.5
##
## $b
## [1] 6.5
##
## $c
## [1] 10.5
```

```r
sapply(l, mean) # "simplifying-list-apply"
```

```
##    a    b    c
##  2.5  6.5 10.5
```

From the tidyverse, `purrr` package adds another way

```r
library(purrr)
map(l, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] 6.5
##
## $c
## [1] 10.5
```

Why use purrr over lapply and sapply?

With `purrr::map()`, inline functions can be easily specified using a formula interface.

```
map(l, ~ . + 2)
```

```
## $a
## [1] 3 4 5 6
##
## $b
## [1]  7  8  9 10
##
## $c
## [1] 11 12 13 14
```

```
map(l, function(x) x + 2)
```

```
## $a
## [1] 3 4 5 6
##
## $b
## [1]  7  8  9 10
```

Other useful map variants

By default, `map()` returns a list like `lapply()`, but we can specify the return type explicity to the simplify the result.

```
map_dbl(l, mean)
```

```
##    a    b    c
##  2.5  6.5 10.5
```

```
map_int(l, length)
```

```
## a b c
## 4 4 4
```

```
map_df(l, mean)
```

```
## # A tibble: 1 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1   2.5   6.5  10.5
```

Other useful map variants (cont'd)

We can also use `map2()` and `pmap()` (and variants) to apply functions over multiple lists.

```
l <- list(a=1:4, b=5:8, c=9:12)
k <- list(d=4:1, e=8:5, f=12:9)
m <- list(rep(1, 4), e=rep(2, 4), f=rep(3, 4))
pmap(list(l, k, m), ~ ..1 + ..2 + ..3)
```

```
## $a
## [1] 6 6 6 6
##
## $b
## [1] 15 15 15 15
##
## $c
## [1] 24 24 24 24
```

```
map2_dbl(l, k, ~ mean(.x + .y))
```

```
##  a  b  c
##  5 13 21
```

Fit models for cross-validation on training sets

Fit models for cross-validation using `purrr::map`:

```
flights_cv <- flights_cv %>%
  mutate(fit = map(train,
                   ~ lm(arr_delay ~ month +
                            dep_time + dep_delay + distance,
                          data = .)))
```

```r
flights_cv
```

```
## # A tibble: 10 x 4
##    train          test           .id   fit
##    <list>         <list>         <chr> <list>
##  1 <S3: resample> <S3: resample> 01    <S3: lm>
##  2 <S3: resample> <S3: resample> 02    <S3: lm>
##  3 <S3: resample> <S3: resample> 03    <S3: lm>
##  4 <S3: resample> <S3: resample> 04    <S3: lm>
##  5 <S3: resample> <S3: resample> 05    <S3: lm>
##  6 <S3: resample> <S3: resample> 06    <S3: lm>
##  7 <S3: resample> <S3: resample> 07    <S3: lm>
##  8 <S3: resample> <S3: resample> 08    <S3: lm>
##  9 <S3: resample> <S3: resample> 09    <S3: lm>
## 10 <S3: resample> <S3: resample> 10    <S3: lm>
```

Get the cross-validated prediction errors

```
flights_cv <- flights_cv %>%
  mutate(rmse_train = map2_dbl(fit, train, ~ rmse(.x, .y)),
         rmse_test = map2_dbl(fit, test, ~ rmse(.x, .y)))
```

```r
select(flights_cv, rmse_train, rmse_test)
```

```
## # A tibble: 10 x 2
##    rmse_train rmse_test
##         <dbl>     <dbl>
##  1       17.7      17.7
##  2       17.7      17.8
##  3       17.7      17.7
##  4       17.7      17.7
##  5       17.7      17.9
##  6       17.7      17.7
##  7       17.7      17.9
##  8       17.7      17.6
##  9       17.7      17.6
## 10       17.7      17.8
```

Cross-validated prediction error

```r
mean(flights_cv$rmse_train) # too optimistic
```

```
## [1] 17.72792
```

```r
mean(flights_cv$rmse_test) # cross-validated error
```

```
## [1] 17.72988
```

Other ways to build models

Besides visualization, how can we decide which variables to add to the model?

Let's start by fitting a separate model for each explanatory variable of interest:

```
fit_month <- lm(arr_delay ~ month, data=flights4)
fit_time <- lm(arr_delay ~ dep_time, data=flights4)
fit_delay <- lm(arr_delay ~ dep_delay, data=flights4)
fit_distance <- lm(arr_delay ~ distance, data=flights4)
```

Which predictor is the best?

```r
rmse(fit_month, flights_part$valid)
```

```
## [1] 42.95449
```

```r
rmse(fit_time, flights_part$valid)
```

```
## [1] 40.24919
```

```r
rmse(fit_delay, flights_part$valid)
```

```
## [1] 17.3306
```

```r
rmse(fit_distance, flights_part$valid)
```

```
## [1] 43.36309
```

Add another variable (p -> 2)

After adding departure delay to the model, what is the next best predictor that improves the model the most?

```
fit_month2 <- lm(arr_delay ~ dep_delay + month, data=flights4)
fit_time2 <- lm(arr_delay ~ dep_delay + dep_time, data=flights4)
fit_distance2 <- lm(arr_delay ~ dep_delay + distance, data=fligh
```

```
rmse(fit_month2, flights_part$valid)
```

```
## [1] 17.15249
```

```
rmse(fit_time2, flights_part$valid)
```

```
## [1] 17.31362
```

```
rmse(fit_distance2, flights_part$valid)
```

```
## [1] 17.25185
```

Add another variable (p -> 3)

After adding month to the model, what is the next best predictor that improves the model the most?

```
fit_time3 <- lm(arr_delay ~ dep_delay + month + dep_time,
                 data=flights4)
fit_distance3 <- lm(arr_delay ~ dep_delay + month + distance,
                data=flights4)
```

```
rmse(fit_time3, flights_part$valid)
```

```
## [1] 17.13132
```

```
rmse(fit_distance3, flights_part$valid)
```

```
## [1] 17.07634
```

Add another variable (p -> 4)

How much does adding the remaining predictor (departure time) improve the model?

```
fit_time4 <- lm(arr_delay ~ dep_delay + month + distance + dep_t
                 data=flights4)
```

```
rmse(fit_distance3, flights_part$valid)
```

```
## [1] 17.07634
```

```
rmse(fit_time4, flights_part$valid)
```

```
## [1] 17.05859
```

Add another variable (p -> 5?)

How much does adding a random variable improve the model?

```
fit_rand5 <- lm(arr_delay ~ dep_delay + month +
                  distance + dep_time + rand,
                  data=flights4)
```

```
rmse(fit_time4, flights_part$valid)
```

```
## [1] 17.05859
```

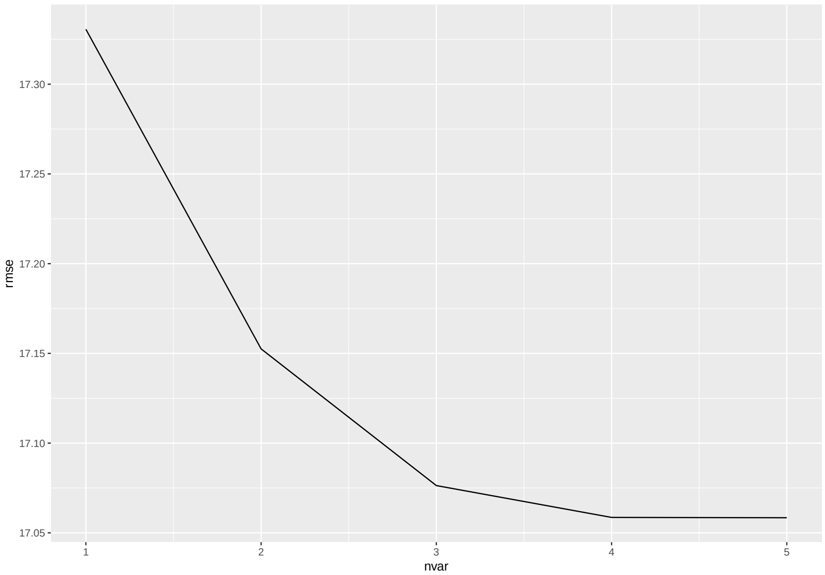
```
rmse(fit_rand5, flights_part$valid)
```

```
## [1] 17.05842
```

How does the model improve as we add variables?

```r
flights5 <- flights_part$valid
fits_rmse <- tibble(nvar = 1:5,
                    rmse = c(rmse(fit_delay, flights5),
                             rmse(fit_month2, flights5),
                             rmse(fit_distance3, flights5),
                             rmse(fit_time4, flights5),
                             rmse(fit_rand5, flights5)))
ggplot(fits_rmse) + geom_line(aes(x=nvar, y=rmse))
```

How does the model improve as we add variables? (cont'd)



Stepwise model selection

Stepwise model selection starts with either:

- No candidate variables in the model
- All candidate variables in the model

Then, each variable is added (forward selection) or dropped (backward elimination) from the model individually.

A selection criterion (e.g., AIC, BIC, RMSE, etc.) is used to determine the optimal variable to add or drop.

Stop the process when adding or dropping a variable would make very little change to the selection criterion.

It is important to evaluate the models on a validation set (and report the final model quality on a test set) to avoid the strong possibility of over-fitting to the training data.

Stepwise model selection in R

```r
step(fit1)
```

```
## Start:  AIC=89778.17
## arr_delay ~ month + dep_time + dep_delay + distance
##
##              Df Sum of Sq      RSS    AIC
## <none>                     4922150  89778
## - dep_time   23     23092  4945241  89805
## - distance    1     41500  4963649  89907
## - month      11     90987  5013137  90042
## - dep_delay   1  23657102 28579252 117193

##
## Call:
## lm(formula = arr_delay ~ month + dep_time + dep_delay + dista
##     data = flights4)
##
## Coefficients:
## (Intercept)       month2       month3       month4       mont
##   -7.972954    -1.425768    -2.838335     1.435123    -5.1916
```