Data Science Practice

STATS 369 Coursebook: Week 4

Lecture 10: Iterations and `map_` Family

Plan for this week

[L10] Iterations and Loops

- Loops with `for`
- `map_` functions

[L11A] Common regression family members 👍

[L11B] Variable selection 👍

- Why?
- How?
- Pros and Cons

[L12] Cross validation

- Motivation
- How?
  - Tuning for model selection
  - Tuning for penalty

Iterations with 'map'

Three basic iterations cases

```r
# applying a function to each element of a list, order does not matter
for (i in 1:npredictors) x[i] <- median(z[,i])

# fixed number of iterations but order matters
for(i in 3:n) fib[i] <- fib[i-1] + fib[i-2] # fibonancii sequence

# order matters, unknown number of iterations
while(n!=1){
  i <- i + 1
  if(n %% 2) n < n/2 else n <- 3*n+1
}
```

What if we want to apply a function?

Sometimes, we apply a function

- to each element of a vector or list
- with the same type of output each time/iteration
- with no memory of previous iterations

Both `map` and `lapply` returns a list.

```
set.seed(765)
lapply(1:3, function(x){round(runif(x),2)})
```

```
## [[1]]
## [1] 0.46
##
## [[2]]
## [1] 0.62 0.76
##
## [[3]]
## [1] 0.75 0.69 0.40
```

```
set.seed(765)
1:3 %>% map(~round(runif(.), 2))
```

```
## [[1]]
## [1] 0.46
##
## [[2]]
## [1] 0.62 0.76
##
## [[3]]
## [1] 0.75 0.69 0.40
```

Why not just loop?

'for' loop

```
set.seed(765)
d.vt <- 1:3
out.ls <- vector('list', length=length(d.vt))
for(i in d.vt){
  out.ls[[i]] <- round(runif(i), 2)
}
out.ls
```

'lapply'

```
set.seed(765)
lapply(1:3, function(x){round(runif(x),2)})
```

'map'

```
set.seed(765)
1:3 %>% map(~round(runif(.), 2))
```

- Less code!
- Distinguishes loops that **need to be looped** from function application. If you have a need/urge to use `for`, think twice.
- Easier to convert to parallel computing.
- `map_dbl`, `map_lgl` etc produce vectors of corresponding kind. There are several other variants that we will get to.
- Despite persistent myths, there is not a significant speed difference in R between writing an explicit loop (e.g. `for` loop) and using `map` or `lapply`.

A toy example

```
# define a function that shuffle n numbers
shuffle <- function(n) sample(1:n, n)
map(3:8, shuffle)
```

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

```
x <- map(1:5, shuffle)  # class(x) #list
x %>% map_dbl(length)
```

```
## [1] 1 2 3 4 5
```

```
x %>% map(length)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5
```

'airquality' Example

```
data(airquality)
airquality %>% map_dbl(mean)
```

```
##      Ozone    Solar.R       Wind       Temp      Month        Day
##         NA         NA  9.957516 77.882353  6.993464 15.803922
```

```
airquality %>% map_dbl(function(x) mean(x, na.rm = TRUE))
```

```
##      Ozone    Solar.R       Wind       Temp      Month        Day
##  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

Grouping

```
cars.df <- read_csv(paste0(base.dir, "/datasets/VehicleYear-2019.csv"))

okcars <- cars.df %>%

  filter(VEHICLE_TYPE == "PASSENGER CAR/VAN") %>%

  filter(BODY_TYPE %in% c("CONVERTIBLE","HATCHBACK","SALOON",
                          "SPORTS CAR","STATION WAGON")) %>%

  filter(BASIC_COLOUR != 'PINK') %>%

  filter(NUMBER_OF_AXLES == 2 & NUMBER_OF_SEATS > 1 &
           NUMBER_OF_SEATS < 8)
```

```
okcars %>%
  group_by(BASIC_COLOUR) %>%
  summarise(mean(GROSS_VEHICLE_MASS))
```

```
## # A tibble: 13 × 2
##    BASIC_COLOUR `mean(GROSS_VEHICLE_MASS)`
##    <chr>                             <dbl>
##  1 BLACK                             2194.
##  2 BLUE                              1953.
##  3 BROWN                             2169.
##  4 CREAM                             2024.
##  5 GOLD                              2128.
##  6 GREEN                             1989.
##  7 GREY                              2161.
##  8 ORANGE                            1849.
##  9 PURPLE                            1804.
## 10 RED                               1954.
## 11 SILVER                            2046.
## 12 WHITE                             2071.
## 13 YELLOW                            1634.
```

```
okcars %>%
  split(.$BASIC_COLOUR) %>%
  map_dbl(~mean(.$GROSS_VEHICLE_MASS))
```

```
##    BLACK     BLUE    BROWN    CREAM     GOLD    GREEN     GREY   ORANGE
## 2193.523 1953.005 2168.940 2023.776 2127.945 1989.059 2161.448 1849.174
##   PURPLE      RED   SILVER    WHITE   YELLOW
## 1803.746 1953.551 2045.851 2070.758 1633.779
```

Notation

- dot (`.`) refers to the data frame we are currently piping
- tilde (`~`) is a cheap way to specify a function,

```
~mean(.$GROSS_VEHICLE_MASS)
```

is short for

```
map_dbl(function(df) mean(df$GROSS_VEHICLE_MASS))
```

Question : what is the difference between `split` and `group_by`?

'split'

For calculating simple statistics such as means, medians etc, `group_by` and `summarise` are simpler and sufficient, but `summarise` can not do everything.

```
models <- okcars %>%
  split(.$BASIC_COLOUR) %>%
  map(~lm(GROSS_VEHICLE_MASS ~ POWER_RATING + BODY_TYPE, data = .))
models[[1]] # the first model, i.e. cohort of black cars
```

```
##
## Call:
## lm(formula = GROSS_VEHICLE_MASS ~ POWER_RATING + BODY_TYPE, data = .)
##
## Coefficients:
##            (Intercept)            POWER_RATING      BODY_TYPEHATCHBACK
##                1071.194                   3.504                 310.629
##        BODY_TYPESALOON     BODY_TYPESPORTS CAR  BODY_TYPESTATION WAGON
##                350.616                -114.224                 726.717
```

Outputs per car colour

```
models %>% map(summary) %>% map_dbl(~.$r.squared) ## r.squared
```

```
##     BLACK      BLUE     BROWN     CREAM      GOLD     GREEN      GREY    ORANGE
## 0.5182092 0.6433944 0.5891971 0.6305280 0.5981567 0.6445379 0.5111112 0.4048807
##    PURPLE       RED    SILVER     WHITE    YELLOW
## 0.6623958 0.6257714 0.6084122 0.5664897 0.3776668
```

```
models %>% map(summary) %>% map_dbl(~round(coef(.)[2],2))  ## beta_1
```

```
##  BLACK   BLUE  BROWN  CREAM   GOLD  GREEN   GREY ORANGE PURPLE    RED SILVER
##   3.50   4.82   6.22   7.54   5.57   5.35   3.98   3.99   7.27   4.30   4.54
##  WHITE YELLOW
##   4.32   2.36
```

Use 'map' to read many files

Here we have a sample of .csv files of Auckland bus locations and delay times.

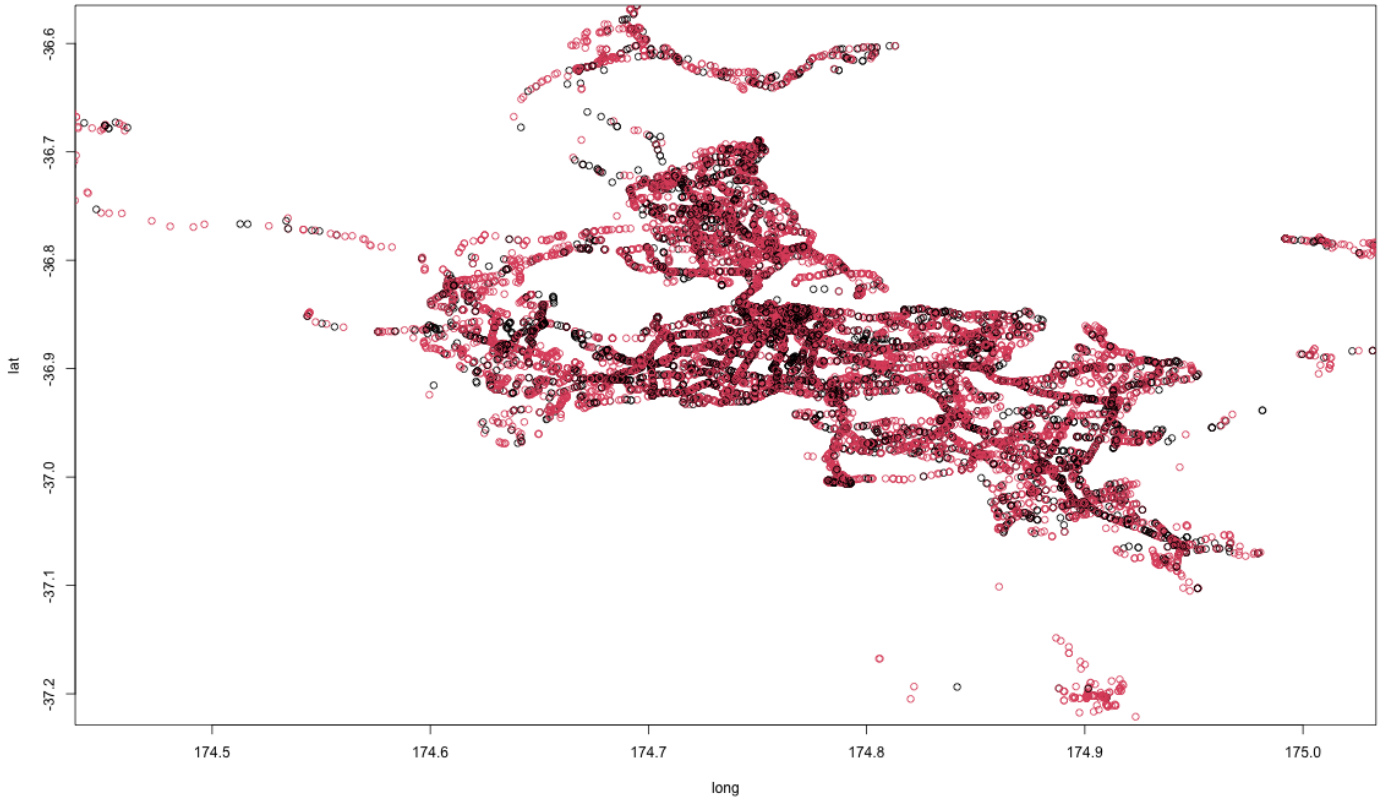
```
bus_files <- list.files(paste0(datasets.dir, "/BUSES"), pattern = "*.csv", full.names = TRUE)
bus_data <- map(bus_files, read_csv)
bus_data %>% map_dbl(nrow)
```

```
##  [1] 428 422 415 429 424 412 427 432 412 427 419 415 422 424 415 433 449 424 450
## [20] 469 472 557 626 729 782 773 711 681 647 638 652 654 652 672 712 711 736 731
## [39] 728 717 694 651 613 596 533 495 443 395 361 506
```

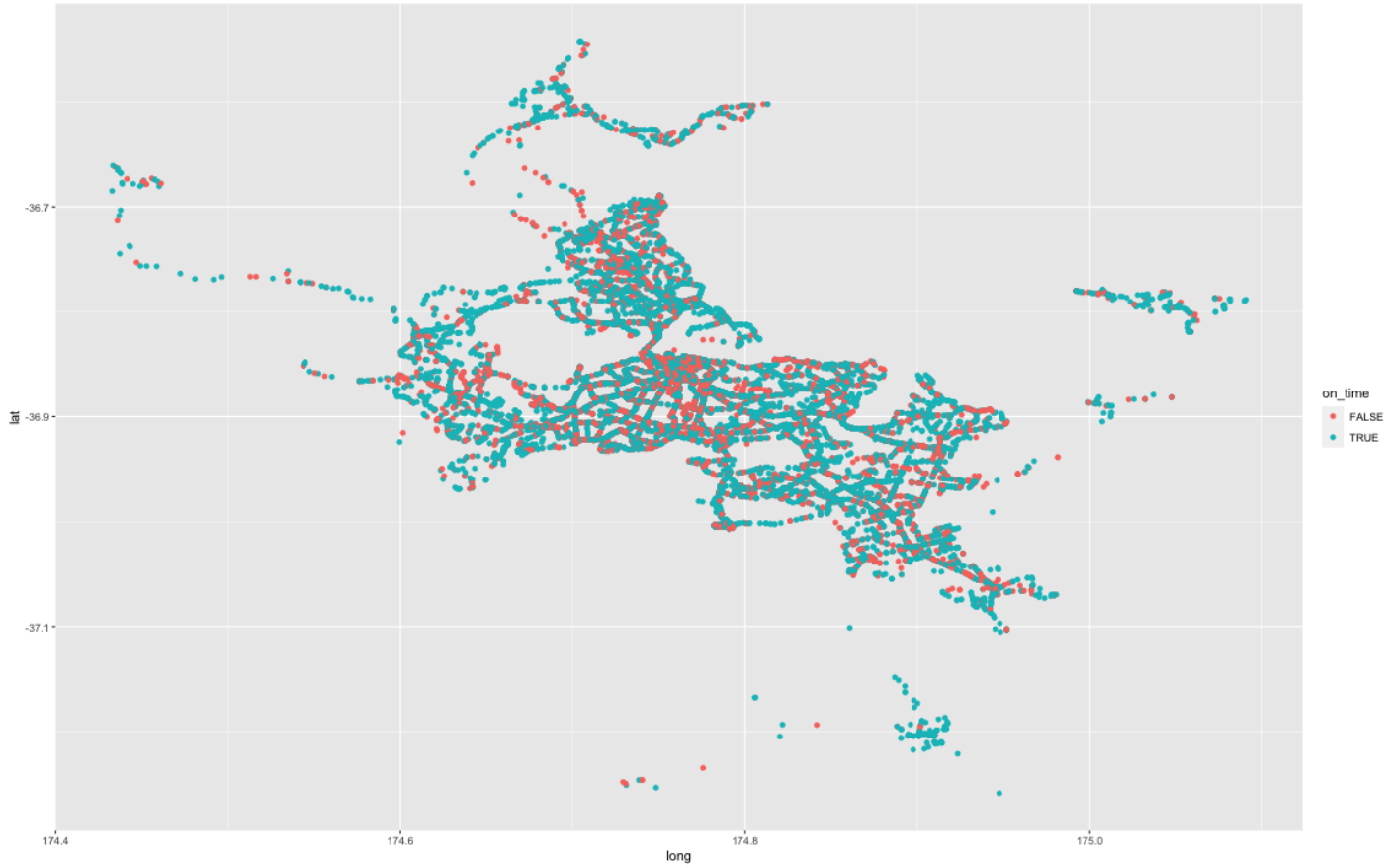
```
bus_data %>%
  map(~mutate(., on_time = (delay > -180) & (delay < 300))) %>%
  map_dbl(~round(mean(.$on_time),2)*100)
```

```
##  [1] 71 73 69 70 72 70 70 75 66 73 74 69 71 70 66 68 70 69 72 68 65 66 65 67 61
## [26] 62 61 63 62 64 61 64 61 65 61 59 56 57 58 59 60 62 60 64 61 67 68 67 69 62
```

```
plot(lat ~ long, data = bus_data[[1]], type = 'n') # empty plot
bus_data %>%
  map(~mutate(., on_time = (delay > -180) & (delay < 300))) %>%
  walk(~points(lat ~ long, data = ., col = .$on_time + 1))
```



```
bus_data %>%
  map(~mutate(., on_time = (delay > -180) & (delay < 300))) %>%
  bind_rows() %>%  # combine all data files by row
  ggplot(data = ., aes(x = long, y = lat, colour = on_time)) +
  geom_point()
```



Lecture 11: Common regression models & variable selection

Plan for this week

[L10] Iterations and Loops

- Loops with `for`
- `map_` functions

[L11A] Common regression family members

[L11B] Variable selection

- Why?
- How?
- Pros and Cons

[L12] Cross validation

- Motivation
- How?
  - Tuning for model selection
  - Tuning for penalty

Common Regression Models

Common regression models --- a perspective from Y

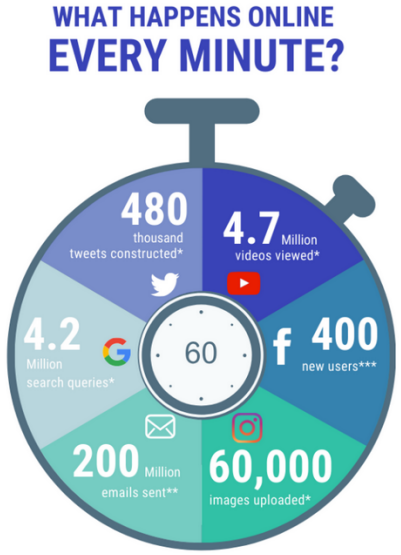
- (Multiple) Linear Regression --- when the response variable Y is *continuous*.
- Logistic regression, Multinomial regression --- when the response variable Y is a *probability* of belonging to a (binary) group (e.g. probability of having a power cut next month).
- Poisson regression, Negative Binomial regression, Quasi Poisson regression --- when the response Y is a *discrete count* (e.g. number of customers arriving at ED).
- Log-normal regression, Cox regression --- when the response variable Y is continuous and strictly *non-negative* (e.g. time-to-event in survival analysis).
- Quantile regression --- when the response variable Y is a *percentile* (e.g. median of house price).
- Ordinal regression --- when the response variable Y is a *rank value* of an ordinal variable (e.g. survey response from scale 1 - 5).
- ...

Note: You can decide which type of regression to use by the characteristic of Y; as for how to fit it, you also need to look at Xs.

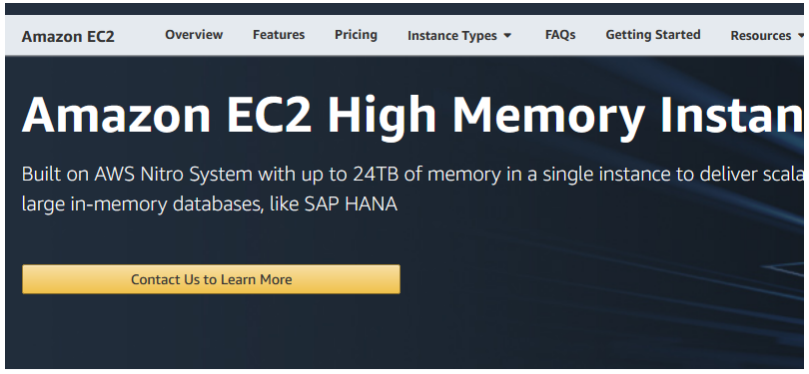
Why Variable Selection?

(or 'feature selection' in ML lingo)

Why do we need variable selection?



> PwC believe that this reached 4.4 ZB in 2019, ... In fact, IDC predicts the world's data will grow to **175 ZB** by 2025! --- sourced by Nodegraph.se



EC2 High Memory instances offer 6, 9, 12, 18, and 24 TB of memory in an instance. These instances are purpose-built to run large in-memory databases, including production deployments of the SAP HANA in-memory database, in the cloud. EC2 High Memory instances allow you to run large in-memory databases and business applications that rely on these databases in the same, shared Amazon Virtual Private Cloud (VPC), reducing the management overhead associated with complex networking and ensuring predictable performance.

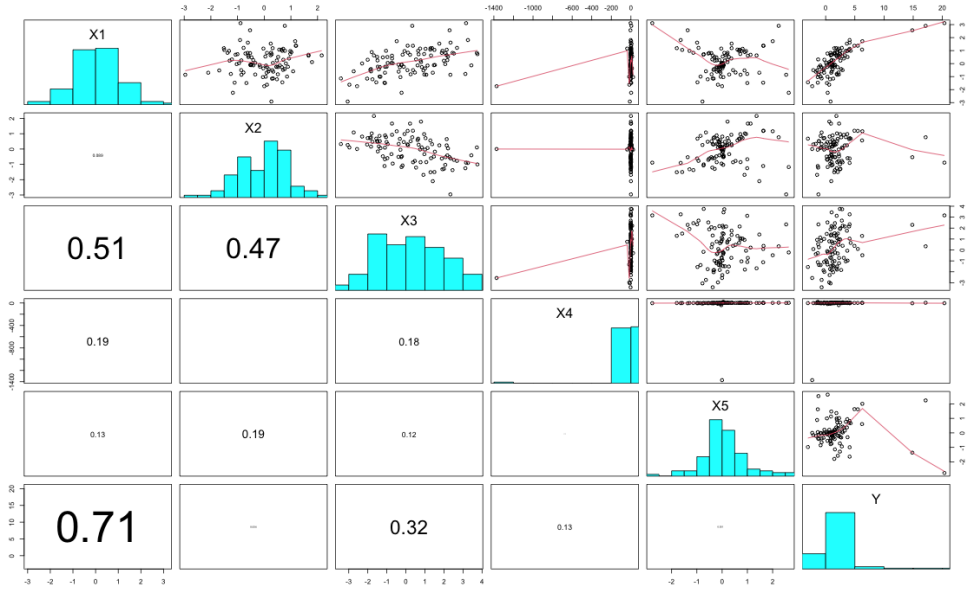
EC2 High Memory instances are EBS-Optimized by default, and offer up to 38 Gbps of dedicated storage bandwidth to encrypted and unencrypted EBS volumes. These instances deliver high networking throughput and low-latency with up to 100 Gbps of aggregate network bandwidth using Amazon Elastic Network Adapter (ENA)-based Enhanced Networking. EC2 High Memory instances with 6 TB, 9 TB, and 12 TB are powered by an 8-socket platform with Intel® Xeon® Platinum 8176M (Skylake) processors. EC2 High Memory instances with 18 TB and 24 TB are the first Amazon EC2 instances powered by an 8-socket platform with 2nd Generation Intel® Xeon® Scalable (Cascade Lake) processors.

Observation: There simply isn't enough RAM to load it all in (and we can have lots of models)!!

Why do we need variable selection?

A toy example

```
set.seed(765)

n = 100
X1 <- rnorm(n)
X2 <- rnorm(n)
X3 <- X1 - X2 + rnorm(n)
X4 <- X1/X2
X5 <- X1*X2

Y <- exp(X1) + sin(X4) + rnorm(n)
```



```
fit1 <- lm(Y~., data=df)
summary(fit1)$coefficients; summary(fit1)$adj.r.square
```

```
##                  Estimate  Std. Error     t value     Pr(>|t|)
## (Intercept)  1.502079e+00 0.242912919  6.18361137 1.609043e-08
## X1           2.537944e+00 0.305083981  8.31883788 6.813268e-13
## X2          -2.784681e-01 0.342253885 -0.81363033 4.179130e-01
## X3          -2.106694e-01 0.208204540 -1.01183875 3.142126e-01
## X4          -8.412643e-05 0.001771045 -0.04750101 9.622147e-01
## X5           1.902070e-01 0.282707379  0.67280513 5.027219e-01

## [1] 0.4923196
```

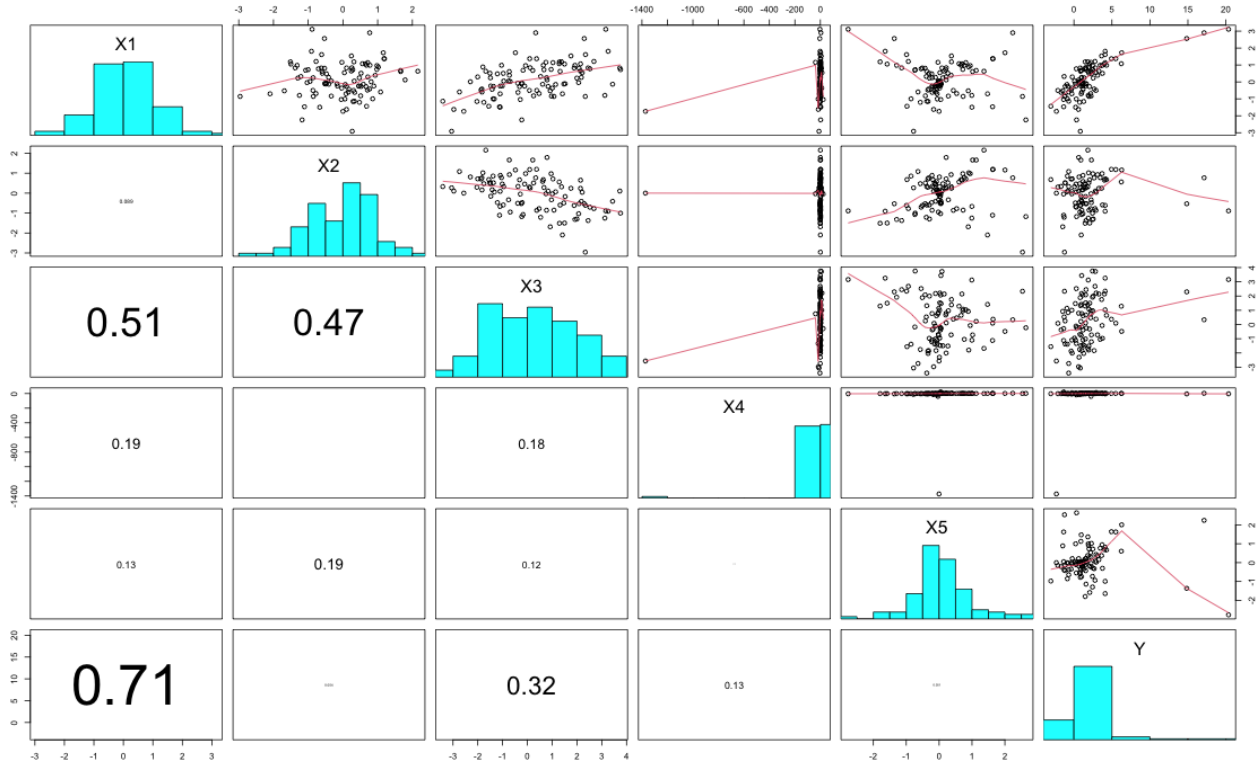
```
fit0 <- lm(Y~X1, data=df)
summary(fit0)$coefficients; summary(fit0)$adj.r.square
```

```
##             Estimate Std. Error   t value     Pr(>|t|)
## (Intercept) 1.509532  0.2350909  6.421056 4.862643e-09
## X1          2.317199  0.2291331 10.112895 6.885156e-17

## [1] 0.5056689
```

Observation: More 'predictors' in a model may *hurt* the model performance.

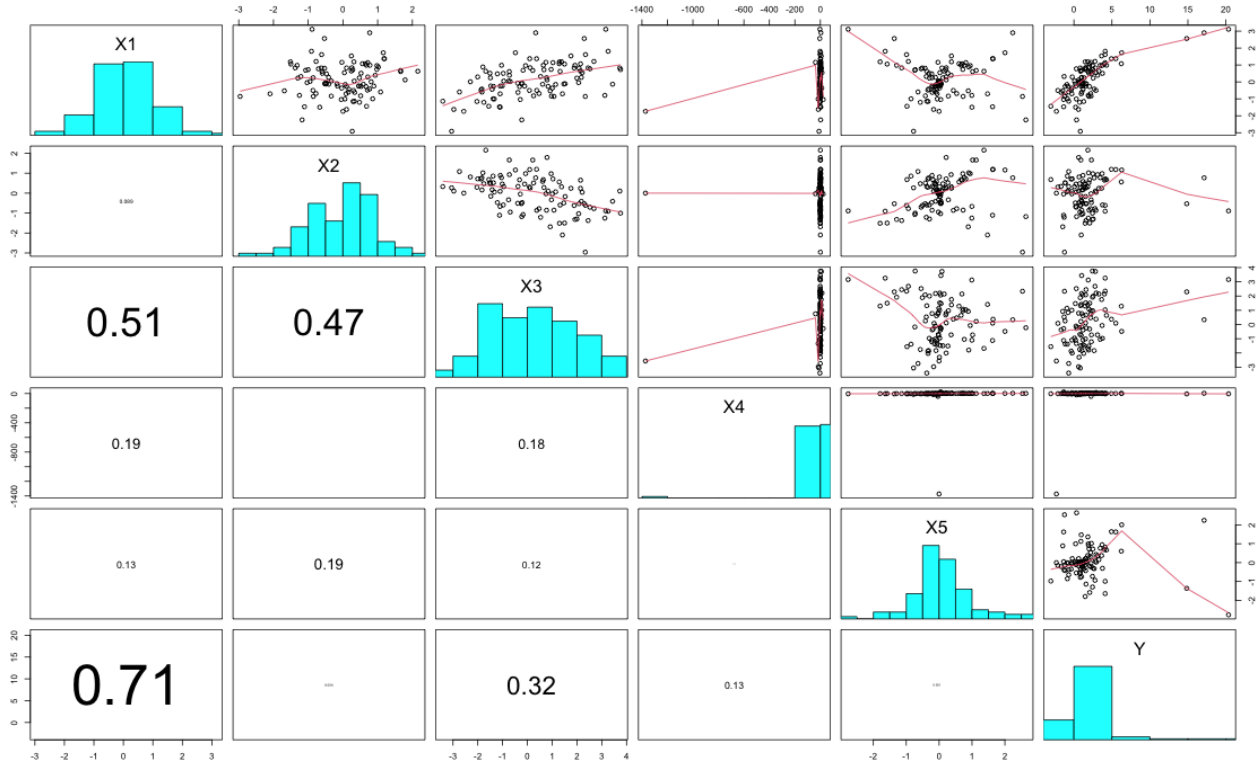
How?

(Intuitively) How?



<u>One intuitive solution:</u> Try *all* possible combinations of Xs and compare performance. (But this is expensive!)

(Intuitively) How?



Another (maybe) intuitive solution: Somehow leverage **correlation**. (How and when to stop?)

Problem -- a more rigorous narrative

We have N observations and P predictors in total. We want to find the set of $p \leq P$ that minimised the *penalised RSS* for a given λ, i.e.

$$\text{Penalised RSS}(\lambda) = nlog(RSS) + \lambda p$$

- There are 2^p possible models -- that is a LOT of (sub) models.
- It is a **non-convex** optimisation problem: a model halfway between two models can be bad, so one cannot just 'head downhill' and hope for the best.
- It is an **NP-hard** problem.
- Until very recently, the best exact algorithm was slow in *most* large examples.

Approximate algorithms

There are three population **approximate** algorithms, i.e, these methods do *not* go through every possible combinations of Xs.

- forward selection
- backwards selection
- stepwise selection

There are feasible for moderate p, and typically gives a model that is not much worse than the best model. [They *can* give a model that *is* much worse than the best model.]

These are **greedy** algorithms: they choose the best model from the small set at each step, without worrying about better future models that they might be discarding. In other words, the algorithm would make locally optimal decision at each step with the assumption it would reach the global optimum in the end.

Forward selection

1. Try all P **one** variable models and pick the best one.
2. Try all $P - 1$ models with one more variable than 1., and pick the best one
3. Repeat until a *stopping rule* is reached or no addition of variable can gain a better fit.

In total, one ends up trying no more than $P \times \texttt{nvmax}$ models, which is feasible.

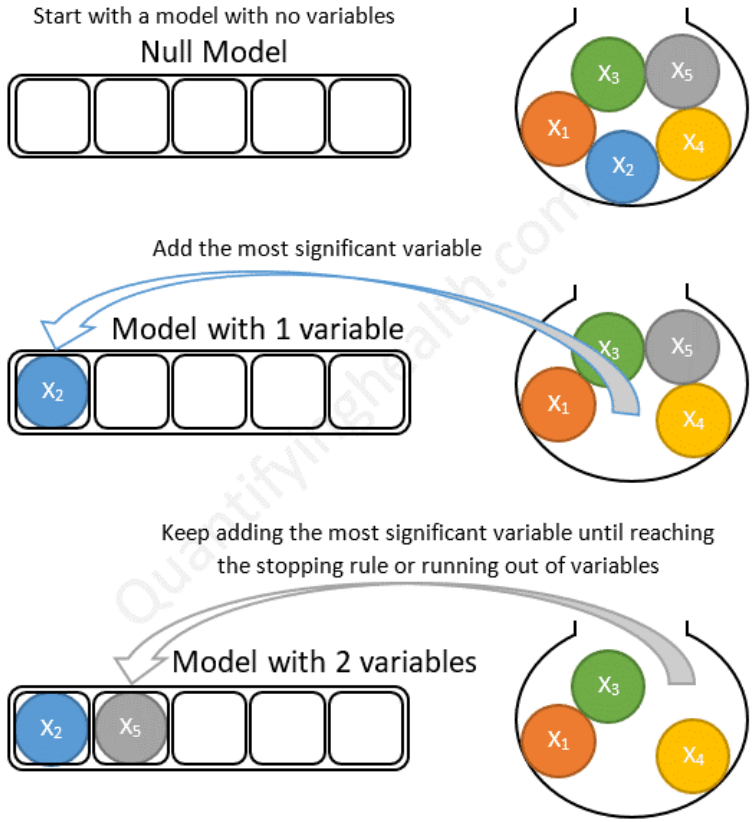
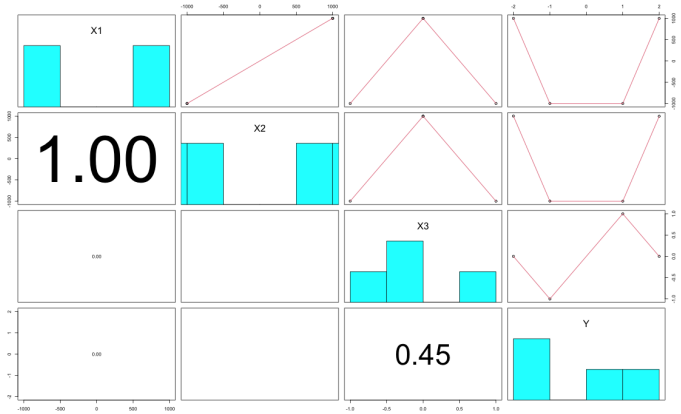


Image source

A minimal example

Suppose we have a small data set like the following, how do we use choose the *best* 2-variable model.

X1	X2	X3	Y
1000	1002	0	-2
-1000	-999	-1	-1
-1000	-1001	1	1
1000	998	0	2



A minimal example

Iteration 1:

```
# get initial correlation
round(cor(min.df), 3)
```

```
##    X1     X2     X3      Y
## X1  1  1.000  0.000  0.000
## X2  1  1.000 -0.001 -0.002
## X3  0 -0.001  1.000  0.447
## Y   0 -0.002  0.447  1.000
```

```
# fit Y~X3, obtain residuals
f1 <- lm(Y~X3, data = min.df)
min2.df <- min.df %>%
  select(-Y) %>%
  mutate(res_f1 = residuals(f1))
```

Iteration 2:

```
# get correlation again
round(cor(min2.df), 3)
```

```
##         X1     X2     X3 res_f1
## X1       1  1.000  0.000  0.000
## X2       1  1.000 -0.001 -0.001
## X3       0 -0.001  1.000  0.000
## res_f1   0 -0.001  0.000  1.000
```

```
# the best 2-var mod:
f2 <- lm(Y~X3 + X2, data=min.df)
```

Note: The 'best' variable at each iteration is the one that gives the largest correlation with the *current* residual.

Backward selection

The backward selection has similar idea except it starts from a full model, i.e. different direction as forward selection.

1. Try the model with ALL P variables
2. Try all P models with $P-1$ variables and pick the best
3. Try all $P-1$ models with one less variable and pick the best
4. Repeat

In total, one ends up trying no more than P^2 models, which is feasible (`nvmax` matters, but only for memory use, not for speed). One would need to be able to fit the model with all variables in the initial place.

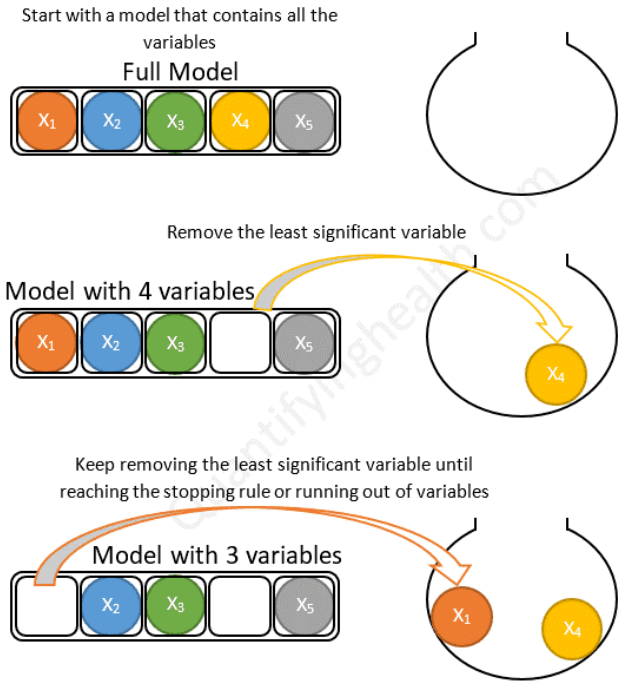


Image source

Stepwise selection

1. Start somewhere
2. Remove variables until it stops helping, i.e. **backward** selection
3. Add variables until it stops helping, i.e. **forward** selection
4. Repeat

This is about the same performance as forward or backwards selection. One can specify a `biggest` model that is smaller than the model with all variables as starting point. (`stepAIC`, or `seqrep` in `regsubsets`).

When to stop?

The stopping rule, say for forward selection, can be simply a model that only contains variables with p-value < threshold.

Common choice of thresholds:

1. A fixed p-value threshold (e.g. 0.05) -- crude, same threshold for all variables!
2. Model with a minimum AIC (Akaike Information Criterion) -- not conservative enough
3. Model with a minimum BIC (Bayesian Information Criterion) -- too conservative sometimes
4. ...

Discussion and Comments

Discussion

Pros:

- Relatively fast and computationally efficient.
- Often serves as a good *starting point to establish a baseline(ish) model.

Cons:

- Too crude (because it is too greedy and use *hard threshold*, i.e. a variable is either 'in' or 'out')
- Forward selection is less effective because 'suppressor variables': it is possible (actually happens in real data) that including X_1 and X_2 reduces the penalised RSS but including either one alone doesn't.
- Backward selection does not have such problem, but need to fit a *big* (e.g. full) model first; unless $N >> P$, the biggest models are NOT well estimated and it can go down the wrong path and get stuck at a sub-optimal model.

The minimal example revisited

Our minimal example earlier has the following data values.

X1	X2	X3	Y
1000	1002	0	-2
-1000	-999	-1	-1
-1000	-1001	1	1
1000	998	0	2

- The forward selection picked X_3 and then X_2. The *true* relationship is $Y = X_1 - X_2$ -- a perfect fit!
- But once the algorithm picked X3, only one of X_1 or X_2 got picked. So, X3 *suppressed* the other two. This is not *uncommon* in real life!
- In this case, if the model includes both X_1 and X_2, then the RSS would be zero; but including either one of them does not outperformed when X_3 is present.

An implementation note: incremental fitting

Adding or removing a single variable from a *linear* model is much faster than fitting the whole model anew.

It is even faster if you just compute the RSS and not the coefficient estimate $\hat{\beta}$.

`regsubsets` uses these tricks (plus the Fortran code) to be fast. `stepAIC()` which fits more general models, can not use these tricks and hence is slow.

Exact algorithms

Exhaustive search: trying all 2^P possible models. Takes about forever. It does not reliably produce better predictive models than forward selection -- increased overfitting vs. better optimisation.

Branch and bound algorithms: the RSS it decreasing and λp is increasing as one adds variables. So one can rule out big branches of the (entire) search tree. This is what `regsubsets` does for exhausive search. Until 2016, it was the best approach.

Model mixed integer-quadratic optimisation (MIO) routines can beat the branch and bound algorithms quite convincingly. There no free-software implementation of the best algorithms, which will slow down general adoption of the methods.

So... what should we use?

No clear clues. Perhaps

- Backward selection if $P << N$
- Forward selection otherwise
- Lasso (content for next week)
- Model MIO software (not yet, but be aware of it.)

A more realistic example

We use mushroom data as an example.

Lecture 12: Cross-validation

Plan for this week

[L10] Iterations and Loops 👍

- Loops with `for`
- `map_` functions

[L11A] Common regression family members 👍

[L11B] Variable selection 👍

- Why?
- How?
- Pros and Cons

[L12] Cross validation

- Motivation
- How?
  - Tuning for model selection
  - Tuning for penalty

Motivation for Cross Validation (CV)

Motivation: Is AIC enough?

- If we have 20 variables, we have $2^{20} \approx 10^6$ possible models
- Most of these models would have lot of variables
- With large-scale model selection, AIC will tend to overfit and prediction will worse
- Even worse for more flexible models (later in the semester)

An example

$n = 100$, $y \sim N(0, 1)$, X is 50 columns of $\overset{iid}{\sim} N(0, 1)$.

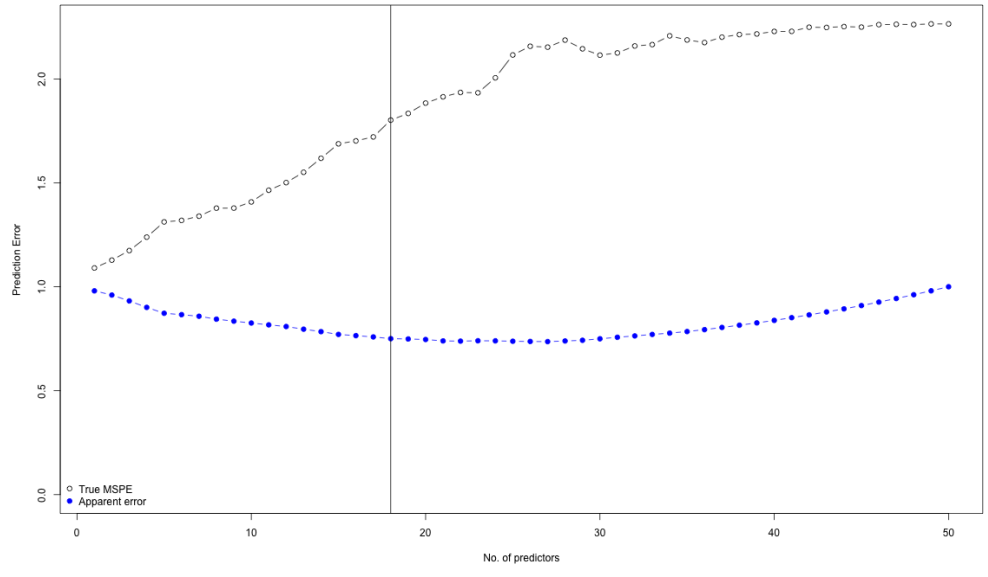
- The best model has no predictors (why?)
- The MSPE of the best model is 1
- The MSPE of any other model is $1 + \sum_1^{50} (\hat{\beta}_i)^2$

We will use `{leap}` package to do backwards stepwise selection. It returns best model of each size.

We will plot the apparent error $\hat{\sigma}^2$, the true MSPE, and show which model minimised AIC.

R code and plot

```
set.seed(765)
y <- rnorm(100)
X <- matrix(rnorm(50*100), ncol = 50)
library(leaps)
sub <- regsubsets(X, y,
                  method = 'back',
                  nvmax = 50)
summ <- summary(sub)
mspe <- sapply(1:50, function(p){
  1 + sum(coef(sub, p)^2)})
aic <- 100 * log(summ$rss) + 2*(1:50)
```



Observations:

- Apparent error tends to go down with more variables, even when true prediction error goes up.
- RSS always goes down but $\hat{\sigma}^2 = \frac{RSS}{n-p}$, so p corrects for part of it.
- AIC is not conservative enough.

Honest estimation of prediction error

We need to

- fit the model on one set of the data (**training data**)
- estimate the prediction error on a separate set (**test data**)

But discarding data from the training set is wasteful. Can we recycle it?

A more generic view

- Recall that the goal for a predictive model is to have low prediction error.
- The apparent error from training data is an *underestimate* of the prediction error for new/unseen data. (See example above.)
- Unless collecting sufficiently large amount of data is feasible and cheap, assessing the predictive model on new data is generally expensive!
- So the question becomes, how can we (re)use the data we have got to *fairly* assess the prediction error? -- **CROSS VALIDATION**.

A bit of history

> Suppose that we **set aside** one individual case, **optimize** for what is left, then **test** on the set-aside case. Repeating this for every case squeezes the data almost dry. If we have to go through the full optimization calculation every time, the extra computation may be hard to face. Occasionally we can easily calculate either exactly or to an adequate approximation what the effect of dropping a specific and very small part of the data will be on the optimized result. This adjusted optimized result can then be compared with the values for the omitted individual. That is, we make one optimization for all the data, followed by one repetition per case of a much simpler calculation, a calculation of the effectof dropping each individual, followed by one test of that individual. When practical, **this approach is attractive.** -- from Mosteller and Turkey (1968)

References:

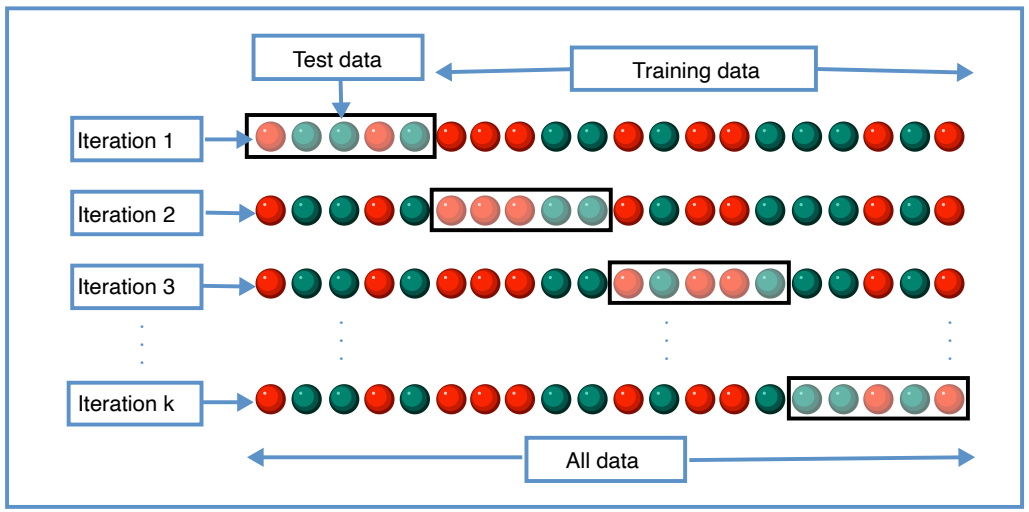
Mosteller, F. and Turkey, J. W. (1968). "Data analysis, including statistics." In Handbook of Social Psychology, Vol2.

Stone, M (1974). "Cross-Validatory Choice and Assessment of Statistical Predictions". Journal of the Royal Statistical Society, Series B (Methodological). 36 (2): 111–147. doi:10.1111/j.2517-6161.1974.tb00994.x

K-fold Cross Validation

$k-$fold cross-validation

K-fold cross validation is one of the commonly used methods for cross validation.

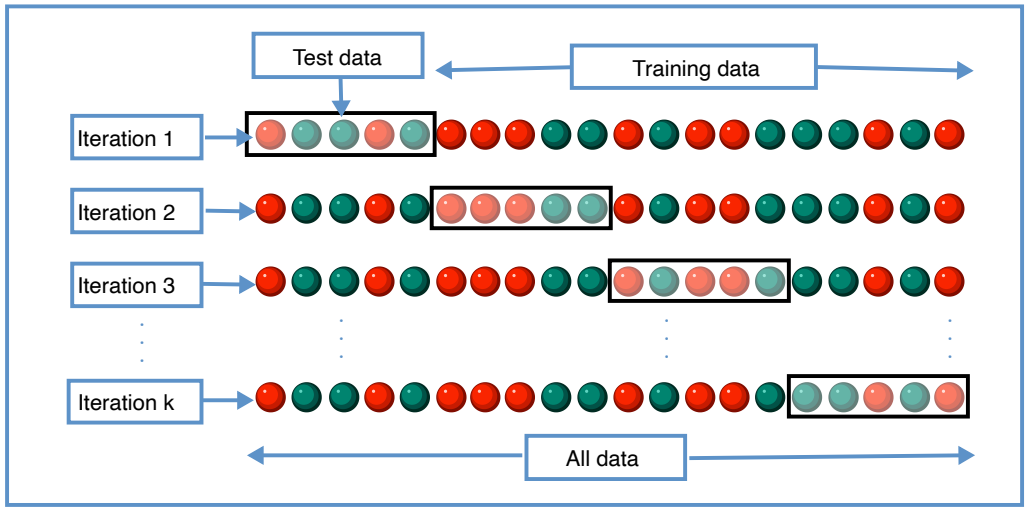


How?

- Divide the data into k subsets (e.g. k = 10, aka '10-folds')
- Fit the model to $k-1$ of them (e.g. train on 9 folds)
- Predict on the kth one, and compute $(Y-\hat{Y})^2$ for each observation in that fold (i.e. test on the remaining fold)
- Repeat (until exhaust all the folds), leaving each fold out in turn.

$k-$fold cross-validation

We now have 1) $(Y - \hat{Y})^2$ for every observation in the (entire) data set; and 2) with Y independent of $\hat{Y}$ for each observation. Hence, $\sum_i^n (Y_i - \hat{Y_i})^2$ should now be an **unbiased** estimator of MSPE.



Comments:

- Efficiency: each data point got used for both training and testing!
- Common choices of K: k=5 and k = 10
- Although not covered in the course, there are many other methods: LOOCV, MC-CV etc

The use of cross validation (CV)

- Case A: Use CV for model selection
- Case B: Use CV for tuning hyper-parameters, such as penalty term

A more rigorous narrative

We want to minimise

$$Penalised\ RSS(\lambda) = nlog(RSS) + \lambda p$$

A: CV on p

- **A1**: run a k-fold CV for each p
- **A2**: run multiple p within each k^{th} fold.

B: CV on λ

- **B1**: run a k-fold CV for each λ.
- **B2**: run multiple λ within each k^{th} fold.

Cross Validation for Model Selection

Use CV for model selection

If we fit the same model in each fold of cross-validation, we get an unbiased estimate of MSPE *for that one model*.

Same effect as using AIC, but more work.

We want to estimate MSPE *not* for a fixed model, but for **whatever model results from applying model fitting strategy to the data.**

What is the model fitting strategy?

When p is not too large...

1. compare all models and pick the one with the smallest AIC.
2. do backwards stepwise search to find the model with the smallest AIC.
3. draw lots of diagnostic plots and think hard about what variables should be in the model (hard to automate!)

General algirhtm for model selection

For each 'fold' from 1 to k:

1. `training` set is the other $k-1$ folds.
2. `test` set is this fold.
3. Run the model fitting strategy on the training set.
4. Predict $\hat{Y}$ for each observation in the test set.

Finally, estimate MSPE as

$$\frac{1}{n}\sum_{i}^{n}(Y_i - \hat{Y}_i)^2$$

But, MSPE of what?

Each fold could end up with a different model! E.g. 5-fold CV will generate 5 models.

What MSPE are we estimating?

The MSPE of a model fitted to these data with this strategy

That is, use the model fitting strategy to fit a model to **all the training data**: the MSPE is an estimate for this model.

Beyond AIC

With large sets of models, the AIC penalty is not conservative enough, and we tend to overfit.

We need some ways to select model size **based on the data**.

Cross-validation lets us estimate MSPE for **model selection strategies:** instead of a fixed strategy (*minimise AIC*), we can have a strategy with a tuning parameter.

For linear regression, a simple strategy is to **'choose the best model with p variables'**, where p is the tuning parameter.

That is, use cross-validation to estimate MSPE for these **strategies**

- choose the best 1-variable model, i.e. $p = 1$
- choose the best 2-variable model, i.e. $p = 2$
- choose the best 3-variable model, i.e. $p = 3$
- etc

Once we know which p minimises MSPE, use that p and fit to the whole data set.

Example (revisited)

Again,

$n = 100$, $y \sim N(0, 1)$, X is 50 columns of $\overset{iid}{\sim} N(0, 1)$.

- The best model has no predictors
- The MSPE of the best model is 1
- The MSPE of any other model is $1 + \sum_1^{50} (\hat{\beta}_i)^2$

We will use `{leap}` package to do backwards stepwise selection. It returns best model of each size.

Setup

```
set.seed(765)
y <- rnorm(100)
X <- matrix(rnorm(50*100), ncol = 50)

library(leaps)
sub <- regsubsets(X, y, method = 'back', nvmax = 50)
summ <- summary(sub)

mspe <- sapply(1:50, function(p) 1 + sum(coef(sub, p)^2))

apparent_error <- summ$rss/(100-(1:50))
```

A1: Cross-validation: one fit

```
yhat <- function(xtrain, ytrain, xtest, p){

  sub <- regsubsets(xtrain,ytrain,nvmax = 50,
                    method = 'back')

  summ <- summary(sub)

  # y_hat = x %*% beta_hat
  beta_hat <- coef(sub, p)
  x_in_mod<-cbind(1,xtest)[, summ$which[p,]]
  y_hat <- x_in_mod %*% beta_hat #

  y_hat
}
```

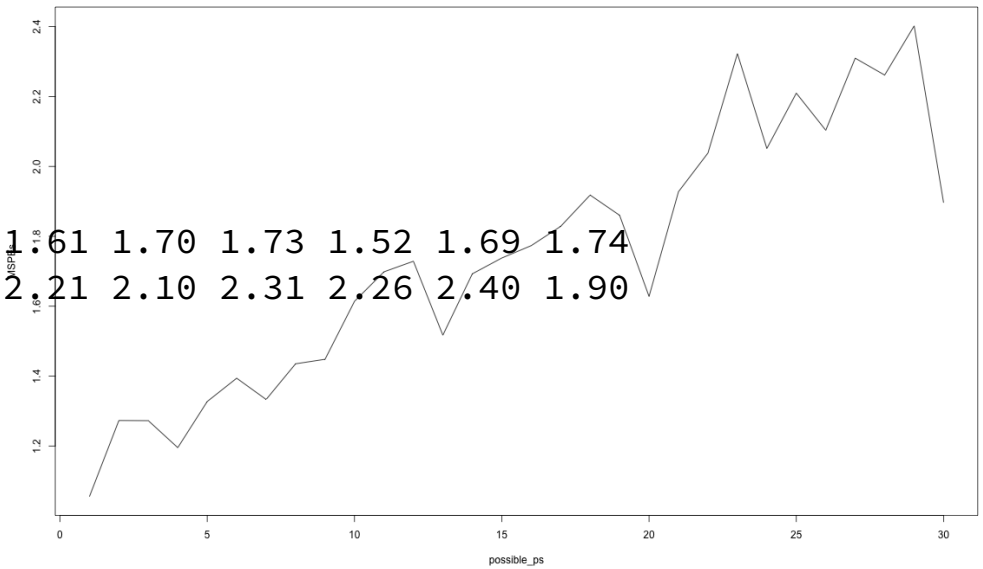
A1: Cross-validation: each p

```
xvalMSPE <- function(p){

  folds <- sample(rep(1:10, 10))  # 10-fold CV

  fitted <- numeric(100)
  for(k in 1:10){
    train <- (1:100)[folds != k]  # train on non-kth folds
    test <- (1:100)[folds == k]  # test on kth folds

    fitted[test] <- yhat(X[train,], y[train], X[test,], p)
  }

  mspe <- mean((y-fitted)^2)

  mspe
}
```

A1: Cross-validation: run it (for multiple p values)

```
set.seed(765)
possible_ps <- 1:30
MSPEs <- map_dbl(possible_ps, xvalMSPE)
round(MSPEs, 2)
```

```
##  [1] 1.06 1.27 1.27 1.20 1.33 1.39 1.33 1.44 1.45 1.61 1.70 1.73 1.52 1.69 1.74
## [16] 1.77 1.83 1.92 1.86 1.63 1.93 2.04 2.32 2.05 2.21 2.10 2.31 2.26 2.40 1.90
```



A1: Fit the model -- with the best p

The best number of variables is 1, so find the best 1-variable model using all the data:

```
best_1var_mod <- regsubsets(X, y, method = 'back', nvmax = 1)
coef(best_1var_mod, 1)
```

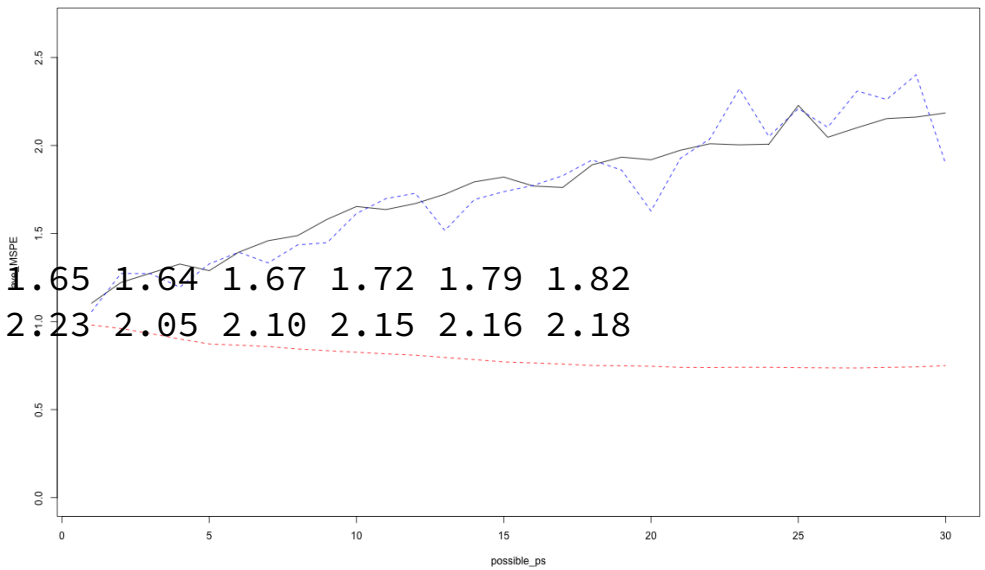
```
## (Intercept)           4
##   0.1249189  -0.2736200
```

A1: Cross validation is noisy

Since cross validation is fast in this example, we might want to average the estimated MSPE over several runs

```
all_MSPEs<-replicate(10,
                     map_dbl(possible_ps,
                             xvalMSPE))

round(rowMeans(all_MSPEs), 2) # average
```

```
##  [1] 1.10 1.22 1.27 1.33 1.29 1.39 1.46 1.49 1.58 1.65 1.64 1.67 1.72 1.79 1.82
## [16] 1.77 1.76 1.89 1.93 1.92 1.97 2.01 2.00 2.01 2.23 2.05 2.10 2.15 2.16 2.18
```



We are picking the right strategy, $p = 1$.

A2: all p within each fold (faster speed)

The code is fairly general, but for linear regression, we can do better.

For a given subset of data, every call to `regsubsets` will return the same set of models. We can do this once, instead of 30 times, by moving the loop over p inside the loop over `fold`.

One p per fold

```
yhat <- function(xtrain, ytrain, xtest, p){

  sub <- regsubsets(xtrain,ytrain,nvmax = 50,
                    method = 'back')

  summ <- summary(sub)

  # y_hat = x %*% beta_hat
  beta_hat <- coef(sub, p)
  x_in_mod<-cbind(1,xtest)[, summ$which[p,]]
  y_hat <- x_in_mod %*% beta_hat #

  y_hat
}
```

All ps per fold

```
allyhat<-function(xtrain, ytrain, xtest,ps){
  yhat <- matrix(nrow=nrow(xtest),
                 ncol=length(ps))
  sub <-regsubsets(xtrain,ytrain, nvmax=50,
                   method="back")
  summ <- summary(sub)
  for(i in seq_along(ps)){
    p<-ps[i]
    betahat<-coef(sub, p)  #<< coefficients
    xinmodel<-cbind(1,xtest)[,summ$which[p,]]
    yhat[,i]<-xinmodel%*%betahat
  }
  yhat
}
```

A2: Loop over folds

```
set.seed(765)
folds <- sample(rep(1:10,10))
possible_ps <- 1:30
fitted <- matrix(nrow = 100, ncol = length(possible_ps))
for(k in 1:10){
  train <- (1:100)[folds != k]
  test <- (1:100)[folds == k]
  fitted[test,] <- allyhat(X[train,], y[train], X[test,], possible_ps)
}
colMeans((y-fitted)^2)
```

```
##  [1] 1.055118 1.133167 1.170449 1.296188 1.333389 1.446132 1.477418 1.496073
##  [9] 1.559642 1.524764 1.591566 1.638741 1.653522 1.667048 1.732656 1.749207
## [17] 1.813618 1.861980 1.847038 1.924343 1.884321 1.940693 1.973002 2.007394
## [25] 2.124749 2.168234 2.209970 2.274876 2.295327 2.395105
```

Not quite *identical*, because previous code samples different `folds` for each p.

Summary

- We can use cross validation to estimate MSPE of *any well-defined model selection strategy*.
- The model selection strategy can have a tuning parameter (hyper-parameter).
- The cross validation estimate of MSPE is approximately unbiased, but noisy (more folds is better, or averaging over repeats with random folds).
- The **whole** model selection strategy needs to be **inside** the cross validation loop, or it is NOT unbiased.

CROSS VALIDATE **ALL THE THINGS**.

Example

The mushroom example (revisited)...

Cross Validation for Parameter Tuning

Using CV for penalty tuning

Using *number of predictors*, p, as the tuning parameter works all right, but

- we may not have any idea what p is good, so we could have try a lot of them
- for methods other than linear regression, there may not be any models with exactly p predictors.

We want something more general.

Cost-complexity penalty

$$\text{Penalised RSS}(\lambda) = nlog(RSS) + \lambda p.$$

AIC has $\lambda = 2$, BIC has $\log(n)$; but can try other λ.

Theoretically, we want $\lambda \geq 2$, since the problem is $\lambda = 2$ is too small (AIC is not conservative enough).

Also, theoretically, λ should not need to be too large: it should increase only as $log(M)$, where M is the number of models with roughly the same true MSPE as the best model.

We can, for example, look at $\lambda = 2, 4, 6, 8, 10$.

Tuning the strategy

For any λ we have a strategy: do backwards stepwise search to find the model with the smallest 'Penalised RSS', i.e. λ.

We know how to estimate $MSPE$ for the model resulting from a strategy, using cross validation.

Meta-strategy:

- for each λ, estimate MSPE for the strategy using λ, to get MSPE for a given λ.
- choose the λ^* that gives the smallest MSPE for a given λ.
- run the strategy on the *whole* dataset using $\lambda = \lambda^*$.

Setup

```
set.seed(765)
y <- rnorm(100)
X <- matrix(rnorm(50*100), ncol = 50)

library(leaps) # for backward selection
sub <- regsubsets(X, y, method = 'back',
                  nvmax = 50)
summ <- summary(sub)

true_mspe <- sapply(1:50, function(p) {
  1 + sum(coef(sub, p)^2)})

apparent_error <- summ$rss/(100-(1:50))
```

B1: General (but insufficient) code

```
yhat <- function(xtrain, ytrain, xtest, lambda){ # lambda is the penalty term for AIC
  sub <- regsubsets(xtrain, ytrain, method = 'back', nvmax = 50)
  summ <- summary(sub)

  # calculate AIC
  aic <- 100*log(summ$rss) + lambda *(1:50)
  best_p <- which.min(aic) # lowest AIC
  beta_hat <- coef(sub, best_p) # coefficient for the 'best' model
  x_in_mod <- cbind(1, xtest)[,summ$which[best_p,]]  # prediction
  y_hat <- x_in_mod %*% beta_hat

}
```

B1: Cross validate

```
xvalMSPE <- function(lambda){
  folds <- sample(rep(1:10,10))
  fitted <- numeric(100)

  for(k in 1:10){  # 10-fold CV
    train <- (1:100)[folds != k]
    test <- (1:100)[folds == k]
    fitted[test] <- yhat(X[train,], y[train], X[test,], lambda)
  }
  mean((y-fitted)^2)
}

lambdas <- (2:10)[c(T,F)] # lambda = 2, 4, 6, 8, 10
map_dbl(lambdas, xvalMSPE)
```

```
## [1] 2.066436 1.445301 1.100878 1.184781 1.183956
```

Looks like $\lambda = 6$ or $\lambda = 10$ is the best. Let us got for $\lambda = 10$ and fit to the *whole* data. *Remember:* larger λ means smaller model.

B1: Results

```
opt_lambda <- 10
sub <- regsubsets(X, y, method = 'back', nvmax=50)
summ <- summary(sub)
aic <- 100*log(summ$rss) + opt_lambda*(1:50)
best_model <- which.min(aic)
coef(sub, best_model)
```

```
## (Intercept)           4
##   0.1249189  -0.2736200
```

Not bad: the best has no predictors, we are only getting one here.

So cross-validation works.

Comments

So tuning p or tuning λ?

- Two approaches are similar in practice
- If you only do linear regression, targeting the number of variables p is easier
- Other methods may be easier with tuning the penalty factor λ.

B2: Faster code for linear regression

- The 'best' model of a given size is *always* the model with smallest RSS - regardless of λ. So we can just do the model search once for each fold, rather than repeating for each λ.
- The code underlying `regsubsets` is already been optimised in terms of speed: leverage lots of linear algebra details and does not actually fit all the models.

B2: Modified 'yhat'

```
allyhat <- function(xtrain, ytrain, xtest, lambdas){

  yhat<-matrix(nrow=nrow(xtest), ncol=length(lambdas))
  sub<-regsubsets(xtrain, ytrain, method="back", nvmax = 50)
  summ<-summary(sub)

  for(i in 1:length(lambdas)){
    penMSE<-100*log(summ$rss)+lambdas[i]*(1:50)
    best_p<-which.min(penMSE) #lowest AIC
    beta_hat<-coef(sub, best_p)#coefficients
    x_in_mod<-cbind(1,xtest)[,summ$which[best_p,]]#predictors in that model
    yhat[,i]<-x_in_mod%*%beta_hat
  }
  yhat
}
```

B2: Cross validate for all λs

```
set.seed(765)
folds<-sample(rep(1:10,10))
lambdas<-(2:10)[c(T,F)]
fitted<-matrix(nrow=100,ncol=length(lambdas))
for(k in 1:10){
  train<-(1:100)[folds!=k]
  test<-(1:100)[folds==k]
  fitted[test,]<-allyhat(X[train,],y[train],X[test,],lambdas)
}
colMeans((y-fitted)^2)
```

```
## [1] 1.849147 1.342222 1.069603 1.055118 1.055118
```

Summary (revisited and beyond)

- We can use cross validation to estimate MSPE of *any well-defined model selection strategy*.
- A very general model selection strategy is to minimise complexity-penalised apparent error, generalising the idea of AIC.
- The cross validation estimate of MSPE is approximately unbiased, but noisy (more folds is better, or averaging over repeats with random folds).
- The **whole** model selection strategy needs to be **inside** the cross validation loop, or it is BIASED.

CROSS VALIDATE **ALL THE THINGS**.

Example

The olive oil example...

Information Leakage

The problem of **information leakage**.

> ... leakage means that information is revealed to the model that gives it an unrealistic advantage to make better predictions. This could happen when test data is leaked into the training set, or when data from the future is leaked to the past. --- from 'Feature Engineering for Machine Learning' (2018)
>
> One other aspect of resampling is related to the concept of information leakage which is where the test set data are used (directly or indirectly) during the training process... In order for any resampling scheme to produce performance estimates that generalize to new data, it must contain all of the steps in the modeling process that could significantly affect the model's effectiveness. --- from 'Feature Engineering and Selection' (2019)

In short, **LOCK AWAY** your test data set all the way until testing!

How can you tell if there is a leakage?

> "too good to be true" performance is "a dead giveaway" of its existence. --- from 'Doing Data Science: Straight Talk from the Frontline' (2013)

Common mistakes

1: Leakage through data transformation

Common mistake: apply data transformation such as standandisation to the *whole* data set *before* randomly split it into training and test sets.

Better practice: bring data preparation into CV step, i.e. for each fold of training and test sets, apply transformation in training set, and use the statistics from training set to transform the test set.

2: Leakage through variable (pre-) selection

Common mistake: pre-select attributes (e.g. by correlations) *before* cross validation and evaluate the models' accuracy by generating CV test errors using *only* the data of the reduced feature set.

Better practice: use a nested cross validation design; i.e. bring the automatic model selection into an inner CV loop, and 'wrap it around' by an outer CV loop for MSPE estimation.

In short, **LOCK AWAY** your test data set all the way until testing!

Resources and References

- Text book 'Introduction to Statistical Learning with Applications in R'
  - Section 3.2: Some Important Questions
  - Section 5.1: Cross Validation
  - Section 6.1: Subset Selection
- Documentation and further examples about `map()` functions can be found from the `{purrr}` package details and Jenny Bryan's tutorial.
- 'Subset Selection in Regression' By Alan Miller (2nd, 2002)
  - Chapter 3 (section 3.1-3.3; 3.7 - 3.8): Finding subsets which fit well.
  - Chapter 5 (section 5.1-5.2; 5.5) : When to stop?
- Cross-validation techniques for model evaluation
- Kaufman,S, Rosset, S and Perlich, C. (2011) *Leakage in Data Mining: Formulation, Dection and Avoidance*. KDD'11, San Diago, California, USA.
- You Might Be Leaking Data Even if You Cross Validation