R General Usage Rubric (CS499/599, Toby Dylan Hocking)

Boldfaced **CH** numbers refer to chapters in *Tao Te Programming* by Patrick Burns.
Preview: https://github.com/tdhock/cs499-599-fall-2020/raw/master/Burns-Tao-Te-Programming.pdf
Buy: https://www.burns-stat.com/documents/books/tao-te-programming/
All of these are common "mistakes" in R coding, please avoid them and you will become a better programmer. **CH2**

- -5 if you create a variable with the same name as any base R function that you are also using in your code. (this is technically possible, but very confusing) solution: use a more informative name that is specific to your application. You can get a list of all base R functions by running the following shell command
  ```
  Rscript --vanilla -e "do.call(c, lapply(search(), ls))"
  ```
  - **BAD**: c <- 0.5, mean <- mean(some.numbers)
  - **GOOD**: decay.constant <- 0.5, cluster.mean <- mean(some.numbers)
- -5 for single-letter variable names, or other un-informative variable names (temp, foo, bar). solution: use an informative (self-documenting) name ideally with a type suffix. **CH31**
  - **BAD**: x <- rnorm(10), temp <- rnorm(10)
  - **GOOD**: random.data.vec <- rnorm(10)
- -5 for variable names and/or identifiers containing numbers/digits where names or another programming technique could be used instead. solution: use informative (self-documenting) variable/identifier names or another programming technique. **CH31**
  - **BAD**: index1, index2, my.list[[6]], my.mat[, 3].
  - **BAD:** my.list <- list(); my.list[[1]] <- data.table(id=1); my.list[[2]] <- data.table(id=2); do.call(rbind, my.list)
  - **BAD:** rbind(data.table(id=1), data.table(id=2))
  - **GOOD**: index.to.keep, index.to.remove, my.list[["squared error"]], my.mat[, "tot.ss"],
- -5 for repeating constants in your code. (this is error-prone, and confusing for readers of your code) solution: declare a variable with that constant and use that variable several times. (the name of the constant should document its purpose/usage) Think: if I wanted to change that constant, in how many places would I need to change it? (the answer should be one) See discussion https://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants and **CH54, CH45**
  - **BAD**: X.train <- X[1:100,]; y.train <- y[1:100]
  - **GOOD**: N.train <- 100; X.train <- X[1:N.train,]; y.train <- y[1:N.train]
  - **BETTER**: train.indices <- 1:100; X.train <- X[train.indices,]; y.train <- y[train.indices]
- -5 for repetitive code blocks / variable names that could be replaced by a loop or another less repetitive programming technique. solution: use for loop and a list with named elements, or a list of data tables, or another programming technique (repetition can and should always be avoided in programming). **CH29, CH4**
  - **BAD repeated numbers of clusters**: result2 <- kmeans(X, 2); result3 <- kmeans(X, 3)
  - **GOOD**: result.list <- list(); for(n.clusters in 2:3) result.list[[paste(n.clusters)]] <- kmeans(X, n.clusters)
  - **BAD repeated set names:** train.err.vec <- computeErr(data.mat[set=="train",])
    valid.err.vec <- computeErr(data.mat[set=="valid",])
    data.table(total.error=c(sum(train.err.vec), sum(valid.err.vec)), set=c("train", "valid"))
  - **GOOD:** data.table(error=computeErr(data.mat), set)[, .(total.error=sum(error)), by=set]
    OR set.err.list <- list(); for(set.name in unique(set)) {
     set.err.list[[set.name]] <- data.table(set, total.error=sum(computeErr(data.mat[set==set.name,]))
    }
    do.call(rbind, set.err.list)
  - **BAD repeated linkage types**: result.complete <- hclust(X, "complete"); result.single <- hclust(X, "single")
  - **GOOD**: result.list <- list(); for(linkage in c("complete", "single")) result.list[[linkage]] <- hclust(X, linkage)
  - **BAD repeated column names**:

```
ARI_kmeans = list()
```

```
for(k in 1:20){
  kc = kmeans(x = data_without_label,centers = k)
  ari = adj.rand.index(kc$cluster,data$V1)
  ARI_kmeans[[k]] = c(k,ari) #K,ARI repetition 1
}
ARI_kmeans = do.call(rbind,ARI_kmeans)
ARI_kmeans = data.frame(ARI_kmeans)
colnames(ARI_kmeans) = c('K','ARI') #K,ARI repetition 2
```

- **GOOD**:

```
kmeans_dt_list = list()
for(k in 1:20){
  kc = kmeans(x = data_without_label,centers = k)
  ari = adj.rand.index(kc$cluster,data$V1)
  kmeans_dt_list[[k]] = data.table(k,ari) #k,ari defined here (no repetition)
}
kmeans_dt = do.call(rbind,kmeans_dt_list)
```

- -5 for lack of consistent indentation. Solution: use consistent indentation which makes it easy to see the structure of your program (for loops, ggplots, etc).
  - **BAD**:

```
some.list <- list()
for(data.i in something){
some.list[[data.i]] <- compute(data.i)
}
```

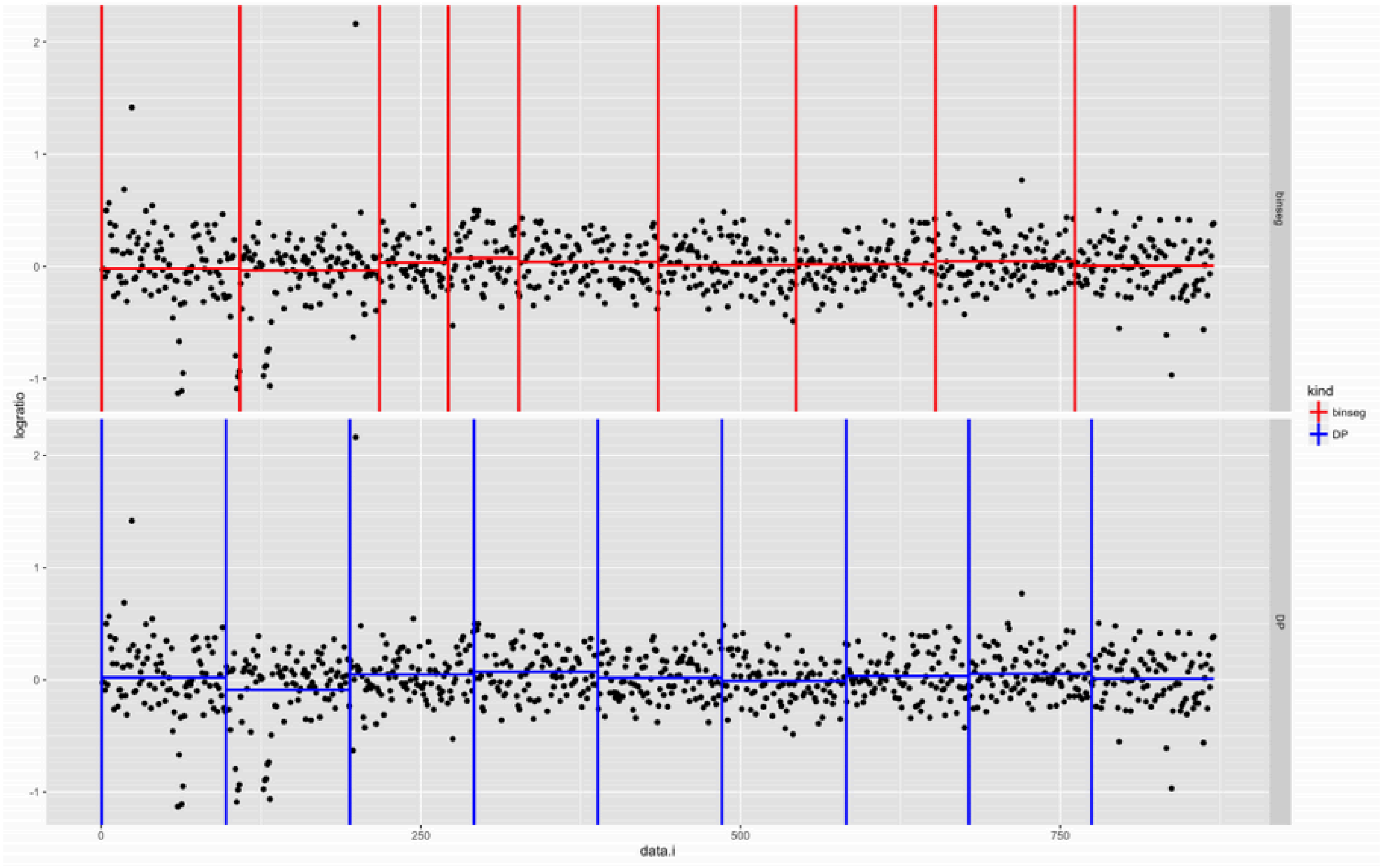
  - **GOOD**:

```
some.list <- list()
for(data.i in something){
  some.list[[data.i]] <- compute(data.i)
}
```

- -5 for using a for loop containing scalar-scalar operations (inefficient) where vector-scalar operations without a for loop could be used (efficient). Not very strict on this one (will only take points away if a for loop is used instead of a more efficient vectorized operation that has already been discussed in class). Please ask google/classmates/myself if you are doing scalar operations inside of a for loop and you don't know of the efficient/good way to do the same thing using vector operations.
  - **BAD/INEFFICIENT**: sum.squares <- 0; for(i in 1:length(x)) sum.squares <- sum.squares + x[i]^2
  - **GOOD/EFFICIENT**: sum.squares <- sum(x^2)
- -5 for using quadratic time accumulation (inefficient). solution: linear time list of data tables (efficient). See discussion https://tdhock.github.io/blog/2017/rbind-inside-outside/ and **CH16, CH47**
  - **BAD/INEFFICIENT**: df <- NULL; for(...) df <- rbind(df, new.df)
  - **GOOD/EFFICIENT**: df.list <- list(); for(...) df.list[[id]] <- new.df; df <- do.call(rbind, df.list)
- -5 for lines of code that are very wide (more than 80 characters) or impossible to read. solution: use intermediate variables or line breaks.

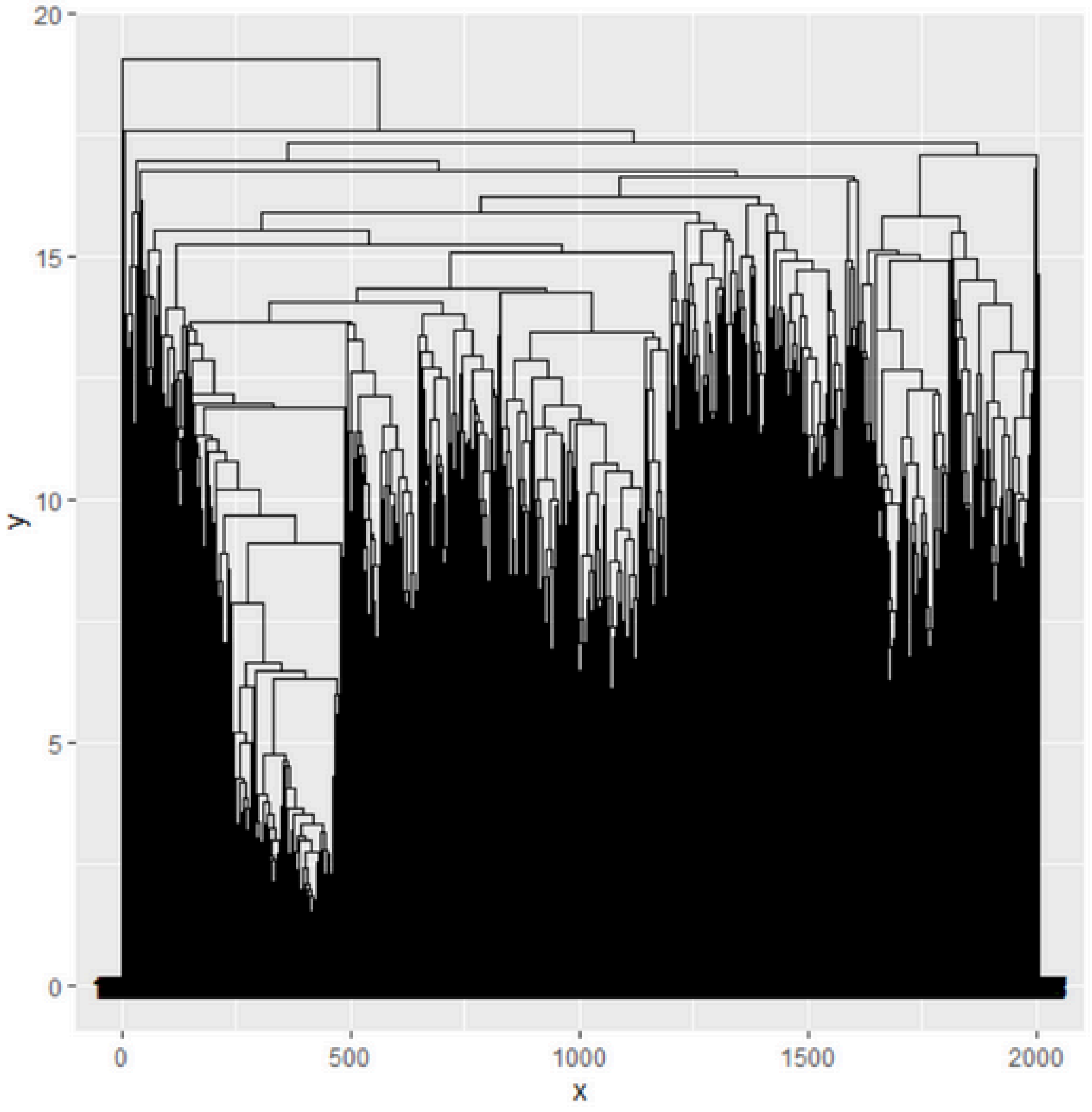
GGPLOT RESULTS

- -5 for plots which are impossible/difficult to read because text is too small or going off plotting area. solution: increase text size and/or plotting area size.
- -5 for using two different visual properties for the same data variable, which is potentially confusing. Solution: only use one visual property per data variable.
  - **BAD**: aes(color=algorithm) + facet_grid(. ~ algorithm), which results in the plot below, potentially confusing because the "kind" data variable is used for both facets and colors.

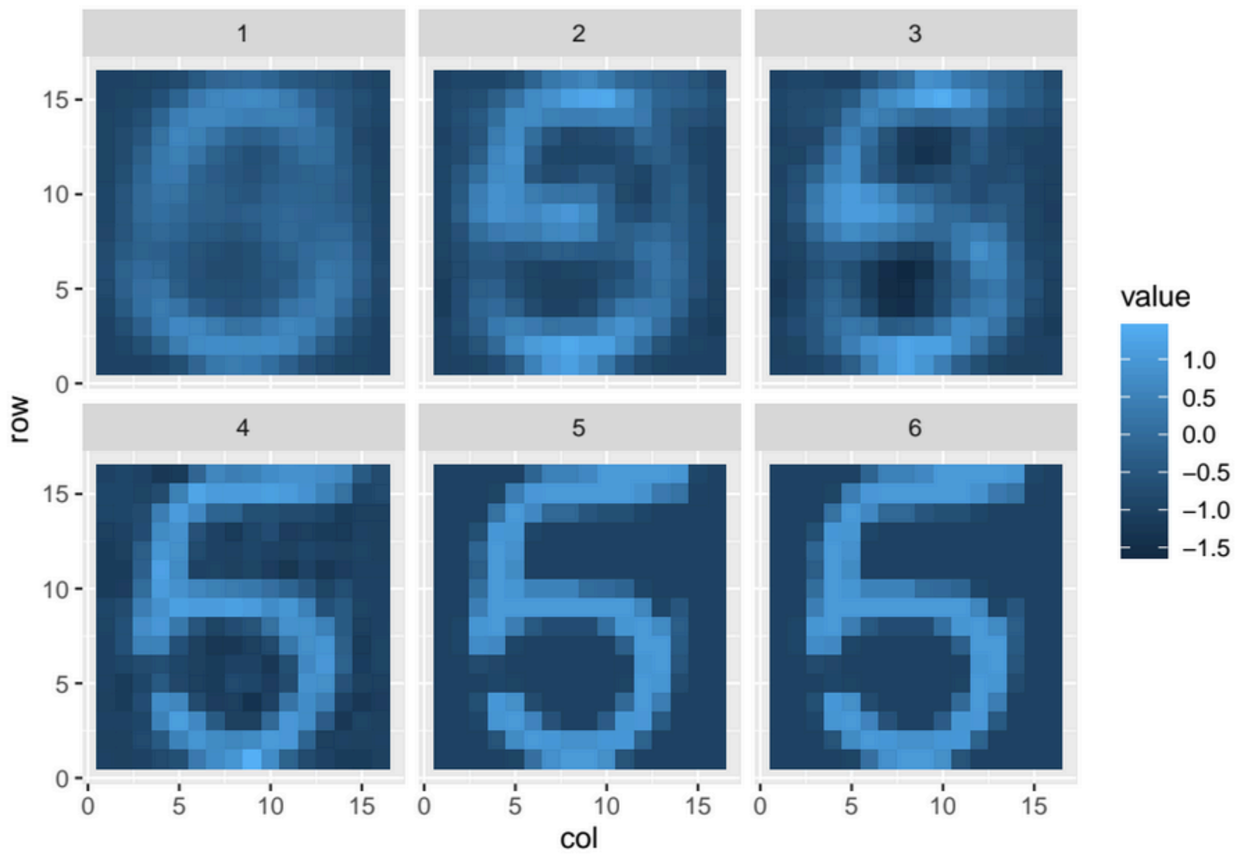


- **GOOD/solution**: use one or the other (not both). In this case it is probably better to remove aes(color) and use facet_grid(labeller=label_both).

- -5 for plots which are impossible/difficult to read for people who are red-green color-blind. solution: do not use red and green on the same figure. See https://daltonlens.org/colorblindness-simulator to simulate what your image would look like to a color blind person, and https://colorbrewer2.org for color palette suggestions (but please avoid the suggestions that have red and green in the same palette).
- -5 for plots which are impossible/difficult to read because of missing/confusing axes/panel labels. solution: make sure there is a title for each axis which explains what it plotted on that axis.
  - Below example confusing because "x" and "y" axes labels do not add any information for the reader. solution: "observation" and "distance" would be good axes labels.

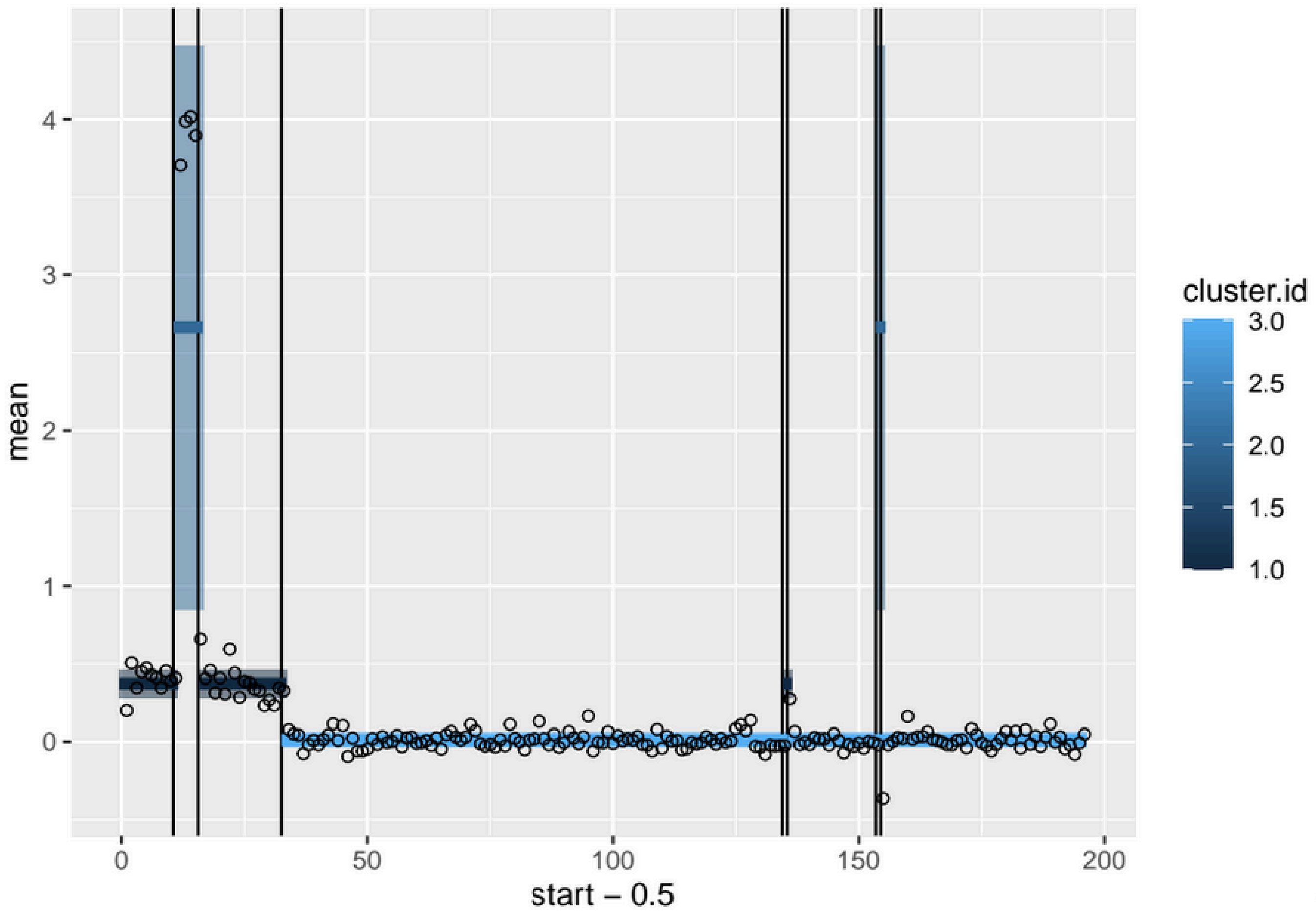


- ○ Below confusing because panel title is number from 1 to 6. Solution: use facet_wrap(labeller=label_both) and more informative values so we get panel titles like "image: 10 PCs" or "image: original.



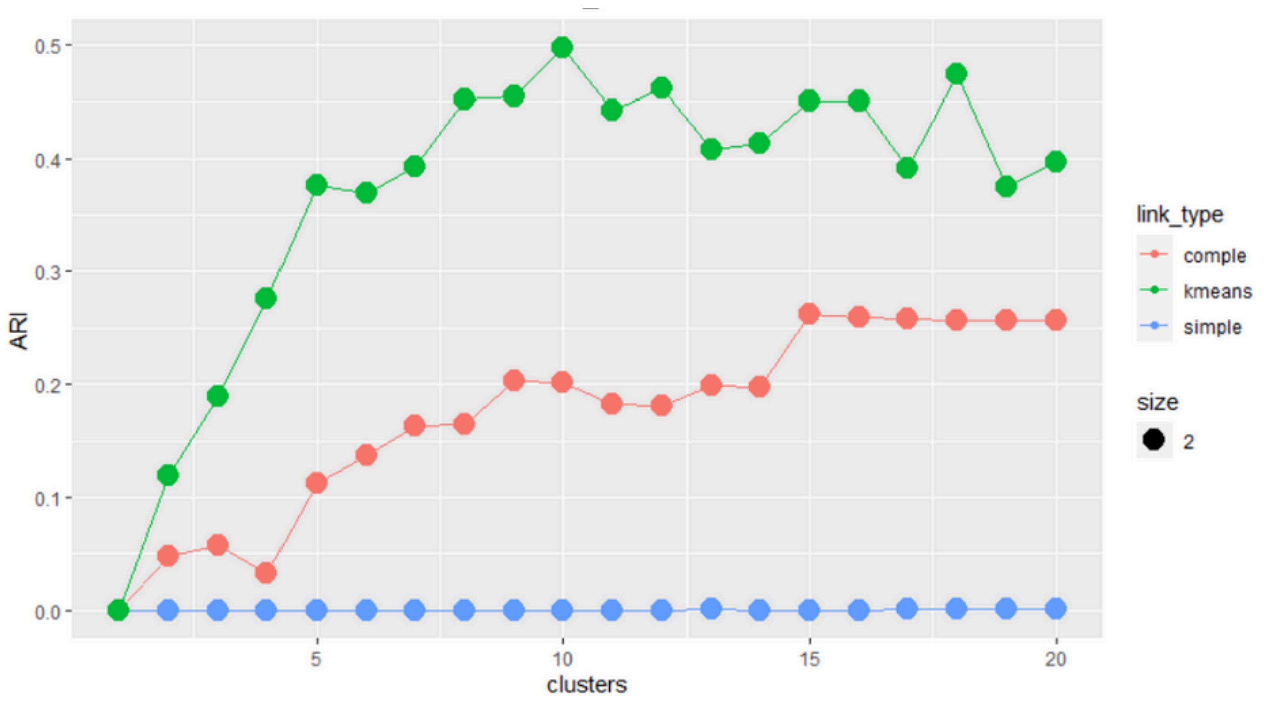
BAD:

- -5 for plots which are impossible/difficult to read because no legend/labels for decoding color/fill/etc. solution: use a legend or direct labels to show what the different colors/etc mean.
- -5 for plots with un-informative / irrelevant /potentially confusing legends. Solution: remove a legend when it does not add any information, or is not relevant to answering the question. Or keep and edit the legend so that it is relevant/informative/not confusing.
    - **BAD example 2**: aes(color=cluster.id) when cluster.id has only 3 values/categories (1, 2, 3), but is type numeric or integer, so ggplot gives you a numerical legend by default, which is confusing because the value 2.5 is shown on the legend, but there is no value 2.5 in the data.



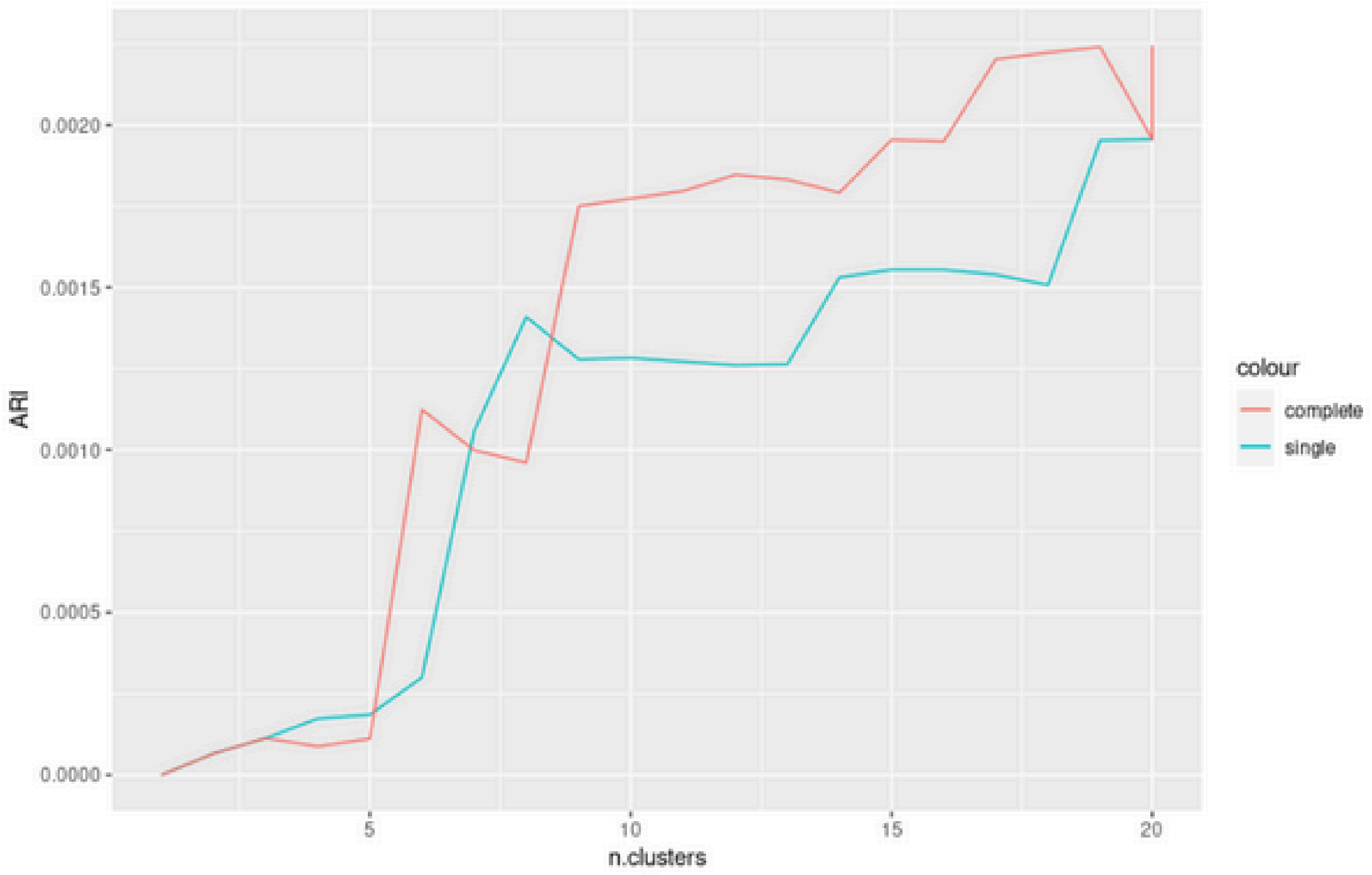
We

   - **GOOD/solution to example 2**: use aes(color=factor(cluster.id)) to get a categorical legend.
- -5 for using constants on the right-hand/value side of aes that will result in a potentially confusing legend with a title such as "colour." solution: if you want a legend then use a data variable instead, with a name that corresponds to what you want showing up on the axis/legend title, like aes(color=some_variable), not aes(color="some constant"). If you don't want a legend, then specify the constant outside of the aes.
   - **BAD example 1**: geom_point(aes(size=2)) gives size legend which has only one value=2 and name "size" so is un-informative.



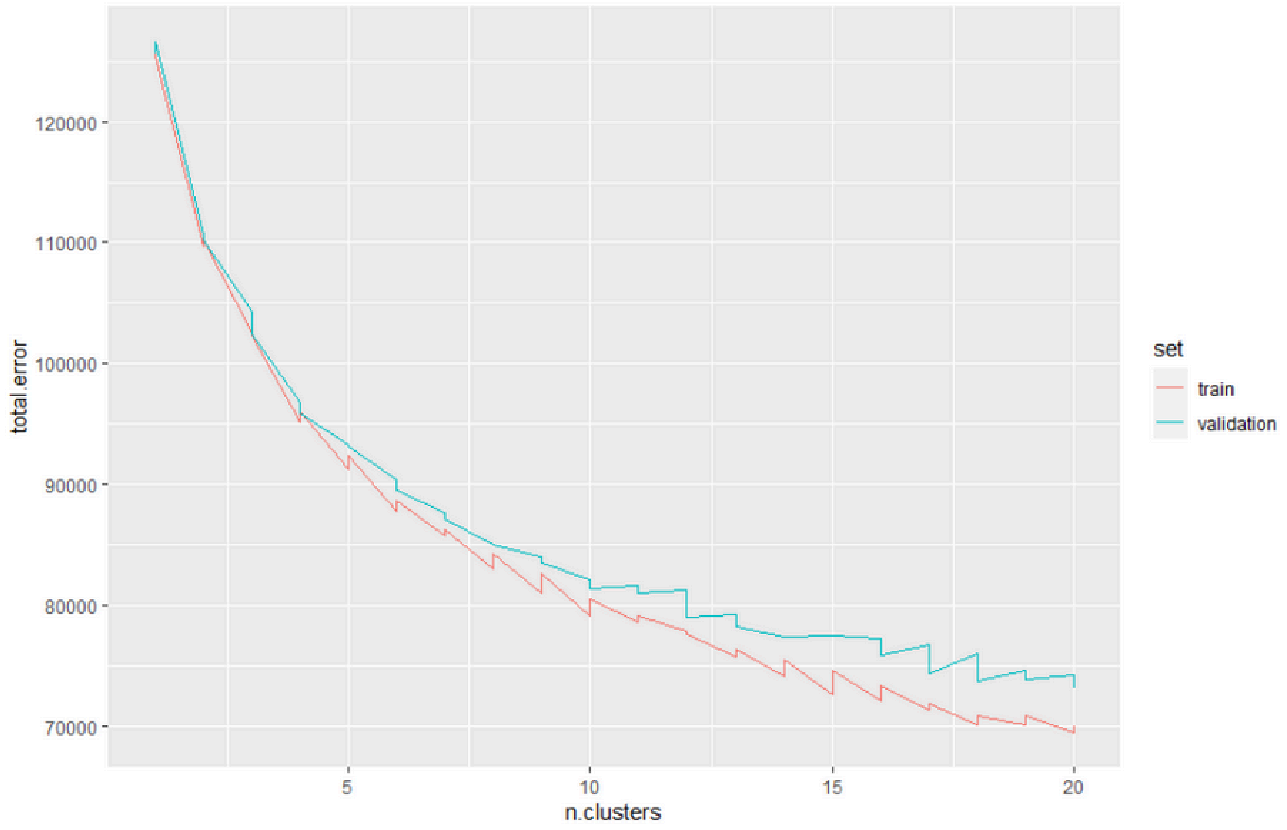
- **GOOD/solution to example 1:** remove that legend by moving size outside the aes, geom_point(size=2).

- **BAD example 2**: geom_point(aes(color="complete")) or geom_point(aes(color="blue")) gives the confusing legend below with title "colour"



- **GOOD/solution to example 2**: geom_point(aes(color=algorithm)) or geom_point(color="blue")

- -5 for missing aes(group) which results in jagged geom_line, with several y values for a single x value. BAD:



Solution: use aes(group=some_variable) so that several different non-jagged lines are drawn instead (each line has one y value per x value).

GGPLOT CODE

- -5 for using multiple geoms in a ggplot when you could use a single geom with a bigger data set. **CH35**
  - **BAD**: geom_line(aes(x, y), data=DT[algorithm=="kmeans"], color="blue")+ geom_line(aes(x, y), data=DT[algorithm=="hclust"], color="red")
  - **GOOD**: geom_line(aes(x, y, color=algorithm), data=DT)+ scale_color_manual(values=c(kmeans="blue", hclust="red"))
- Other common mistakes may appear here in the future as needed.

Other suggestions

- **stringr::str_match** (double colon package notation) is preferable to **library(stringr)**, so readers of your code can easily see in what package each function you use is defined.
- **setwd("C:/...")** with an absolute path should be avoided. Relative paths are more portable (I could potentially run the code without modification on my computer)
- **return(something)** at the end of a function is un-necessary in R (potentially confusing). **return(something)** is ok for early return, but for the end of the function just use **something** (no return)
- **dataset <- some_fun(dataset)** is potentially confusing / error-prone because dataset means one thing before that line of code, and something else after. It is better to give unique variable names (avoid over-writing the same varaible with another value), GOOD: **transformed_data <- some_fun(original_data)**
- The base pipe |> (always available in recent R versions) should be preferred to the magrittr pipe %>%

Java General Usage Rubric (CS249, Michael Leverington, not used for grading CS499/599, but good guidelines to learn/follow)

1. use of non-self-documenting or single-letter variable: -1 per declaration or use
2. missing or non-aligned curly braces: -1 per pair/occasion
3. use of unnecessary language prepends (e.g., java.lang. or SomeNodeType.): -1 per occasion
4. redundant boolean test: -1 per occasion
   a. e.g., **`if( <boolean expression> == false )`**
   b. e.g., **`if( <boolean expression> == true )`**
5. second, or subsequent, **`if`** statement that should logically be **`else`**: -2 per occasion
6. any data/state change in array brackets or in method parameters: -2 per occasion
   a. e.g., **`value = array[ index++ ];`**
   b. e.g., **`value = someMethod( otherValue, myValue-- );`**
   c. note: **`array[ index + 1 ]`** or **`someMethod( otherValue, myValue – 1 )`** are appropriate and acceptable
7. declaration of variable within any loop: -2 per occasion
   a. e.g., **`for( int index = 0; ... )`**
   b. e.g.,
   ```
   while( someCondition )
             {
              int newValue = someValue;
              ...
   ```
8. use of incomplete **`for`** loop (e.g., **`for(   ; index < 5; index++ )`**): -2 per occasion
9. use of **`if/else`** in place of single Boolean **`return`** statement: -2 per occasion
10. any code on the same line as a curly brace: -2 per occasion
11. use of 1 or 0 (or any other numbers) in place of **`true`** or **`false`**: -2 per occasion
12. use of literals (e.g., **`int`**s, **`char`**s, etc.) when constants or variables should be used: -2 per occasion
    a. Note that in most cases, **`String`** literals are more readable than constants or variables so **`String`** literals are recommended unless the same **`String`** value is used in several places
13. use of **`break`** anywhere but in a **`switch`** operation: -2 per occasion
14. use of unspecified **`try/catch`**: -2 per occasion
15. use of **`return;`** (without value): -3
16. use of **`continue`** anywhere: -3 per occasion
17. any I/O in a method not specified for I/O operations: -3 per occasion
18. use of methods in parameter/argument lists: -3 per occasion
19. use of methods in array brackets: -3 per occasion
20. creation or use of empty **`if`** or **`else`** blocks: -3 per occasion
21. placement of **`main`** method in any ADT class: -3 per occasion
22. code more than 80 characters: -3 per five lines
23. use of "var" for data typing: -3 per occasion
24. use of ternary operator: -3 per occasion
25. use of unspecified Java or other utility methods and/or data: -5 per occasion
26. --- new items will be added as needed ---