Repeating Tasks with `purrr::map` and `purrr::pmap`

Introduction

A Common Structure

We often want to:

- Perform the same task a bunch of times with slight variations
- Assemble the results so that we can use them later

Example: Get KNN Predictions with Different Values of k

- For each value of $k = 1, \ldots, 100$, get predictions from a KNN fit using that number of neighbors.
- Assemble the results into a data frame

Later, we plan to plot the predictions in that data frame.

The `map`/`pmap` Approach:

- Write a function that does the task we want to repeat
  - It takes one (`map`) or more (`pmap`) arguments that specify what is done differently each time we do the task
- Create a vector or data frame of the arguments that will be used
- Use a function in the `map` family to call our function with every possible argument and collect the results together

Table of `map` and `pmap` functions

Number of Function Arguments that Change	Results Should Be Assembled Into...	Function to Use
1	list	`map`
	data frame (stacked vertically, rows)	`map_dfr`
	double/numeric vector (-1, 7.132, 0.001)	`map_dbl`
	logical vector (TRUE/FALSE)	`map_lgl`
	integer vector (-1, -5, 9, 0)	`map_int`
	character vector ("kittens", "cats")	`map_chr`
	data frame (stacked horizontally, columns)	`map_dfc`
≥ 2	list	`pmap`
	data frame (stacked vertically, rows)	`pmap_dfr`
	double/numeric vector (-1, 7.132, 0.001)	`pmap_dbl`
	logical vector (TRUE/FALSE)	`pmap_lgl`
	integer vector (-1, -5, 9, 0)	`pmap_int`
	character vector ("kittens", "cats")	`pmap_chr`
	data frame (stacked horizontally, columns)	`pmap_dfc`

In my experience, the most useful output types tend to be lists, double vectors, and data frames (stacked vertically).

Examples

Let's look at examples from previous classes.

```
library(readr)
library(dplyr)
library(ggplot2)
library(gridExtra)
library(purrr)
library(MASS)
```

`map`: Your function has only 1 argument that varies

Example 1: Assemble output into a data frame. `map_dfr`

We will get KNN regression predictions on a grid of points, assemble into a data frame for later plotting. (This code adapted from the KNN Regression handout on 2018-09-24.)

```
# set up grid of values to get predictions
lstat_grid <- seq(from = min(Boston$lstat), to = max(Boston$lstat), by = 0.1)
test_data <- data.frame(
  lstat = lstat_grid
)

# Function we will call multiple times
# Note that it returns a data frame!
get_test_preds_df_k <- function(k) {
  data.frame(
    lstat = lstat_grid,
    medv_hat = knn.reg(
      train = Boston %>% dplyr::select(lstat),
      test = test_data,
      y = Boston %>% dplyr::select(medv),
      k = k)$pred,
    k = k
  )
}

# Example of calling the function once
temp <- get_test_preds_df_k(k = 5)
str(temp)
```

```
## 'data.frame':    363 obs. of  3 variables:
##  $ lstat   : num  1.73 1.83 1.93 2.03 2.13 2.23 2.33 2.43 2.53 2.63 ...
##  $ medv_hat: num  42.6 42.6 42.6 42.6 42.6 ...
##  $ k       : num  5 5 5 5 5 5 5 5 5 5 ...
```

```
head(temp)
```

```
##   lstat medv_hat k
## 1  1.73     42.6 5
## 2  1.83     42.6 5
## 3  1.93     42.6 5
## 4  2.03     42.6 5
## 5  2.13     42.6 5
## 6  2.23     42.6 5
```

```
temp %>% distinct(k)
```

```
##   k
```

```
## 1 5
```

```
# Using map_dfr to call get_test_preds_df_k multiple times and assemble the results
#  * The first argument to map_dfr says what values to use when calling get_test_preds_df_k
#    Here we will call get_test_preds_df_k 6 different times with 6 different values of k.
#  * The second argument to map_dfr is the function to call, get_test_preds_df_k
#  * Because we used the version of map ending in _dfr, the results from each call to
#    get_test_preds_df_k are stacked vertically to create one big data frame
test_boston <- map_dfr(
  c(1, 5, 10, 50, 100, nrow(Boston)),
  get_test_preds_df_k
)
str(test_boston)
```

```
## 'data.frame':    2178 obs. of  3 variables:
##  $ lstat   : num  1.73 1.83 1.93 2.03 2.13 2.23 2.33 2.43 2.53 2.63 ...
##  $ medv_hat: num  50 50 50 34.9 34.9 41.7 41.7 41.7 41.7 41.7 ...
##  $ k       : num  1 1 1 1 1 1 1 1 1 1 ...
```

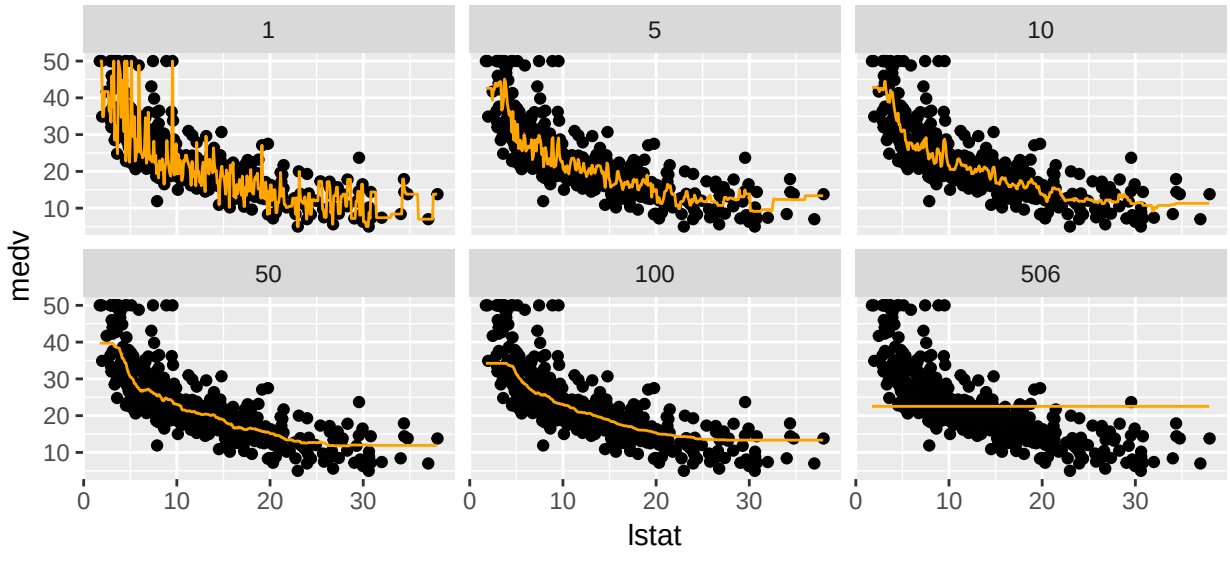
```
head(test_boston)
```

```
##   lstat medv_hat k
## 1  1.73     50.0 1
## 2  1.83     50.0 1
## 3  1.93     50.0 1
## 4  2.03     34.9 1
## 5  2.13     34.9 1
## 6  2.23     41.7 1
```

```
test_boston %>% distinct(k)
```

```
##     k
## 1   1
## 2   5
## 3  10
## 4  50
## 5 100
## 6 506
```

```
# Make a plot.
ggplot() +
  geom_point(data = Boston, mapping = aes(x = lstat, y = medv)) +
  geom_line(data = test_boston, mapping = aes(x = lstat, y = medv_hat), color = "orange") +
  facet_wrap( ~ k)
```



Example 2: Assemble output into a list. `map`

This code adapted from the KNN Classification handout from 2018-10-10.

```
library(ISLR)

# Intial rescaling of explanatory variables
Default <- Default %>%
  mutate(
    balance_rescaled = balance / sd(balance),
    income_rescaled = income / sd(income)
  )

# Set up grid of values to get predictions
max_balance_rescaled <- max(Default$balance_rescaled)
max_income_rescaled <- max(Default$income_rescaled)
background <- expand.grid(
    balance_rescaled = seq(from = 0, to = max_balance_rescaled, length = 101),
    income_rescaled = seq(from = 0, to = max_income_rescaled, length = 101))

# Function we will call multiple times
# It returns a ggplot object to plot later!
get_knn_plot <- function(k) {
  knn_predictions <- knn(
    train = Default %>% dplyr::select(balance_rescaled, income_rescaled),
    test = background,
    cl = Default$default,
    k = k,
    prob = TRUE)

  background_knn <- background %>%
    mutate(
      est_default = knn_predictions
    )

  p_k <- ggplot() +
    geom_point(data = background_knn,
      mapping = aes(x = balance_rescaled, y = income_rescaled, color = est_default), size = 0.1, alpha = 0.5)
    geom_point(data = Default, mapping = aes(x = balance_rescaled, y = income_rescaled, color = default)) +
    scale_color_discrete("Default") +
    geom_abline(intercept = 1.154e+01 / 2.081e-05, slope = - 5.647e-03 / 2.081e-05) +
    ggtitle(paste0("KNN, k = ", k))

  return(p_k)
}

# Using map to call get_knn_plot multiple times and assemble the results.
#  * The first argument says what values to use when calling get_knn_plot.
#    Here we will call get_knn_plot 4 times, with four different values of k.
#  * Because we used map (without anything like _dfr or _dbl on the end),
#    results of these calls will be assembled into a list!!
plots_knn <- map(
  c(1, 10, 100, 1000),
  get_knn_plot
)

# look at what the results of the call to map look like.
typeof(plots_knn)
```

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

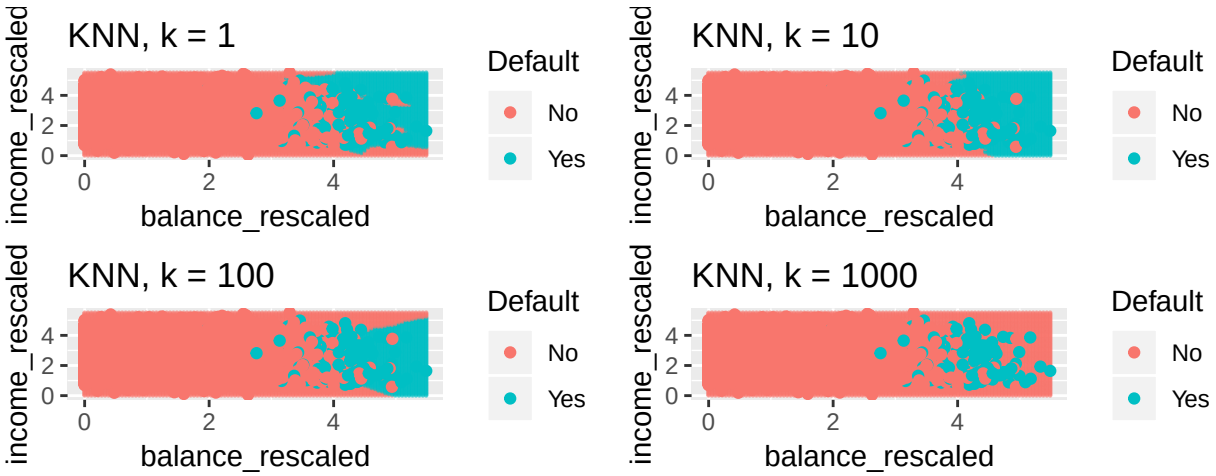
```
length(plots_knn)
```

```
## [1] 4
```

```
class(plots_knn[[1]])
```

```
## [1] "gg"     "ggplot"
```

```
# Print out the plots we made.
grid.arrange(plots_knn[[1]], plots_knn[[2]], plots_knn[[3]], plots_knn[[4]], ncol = 2)
```



Example 3: Assemble output into a numeric vector (of type double). `map_dbl`

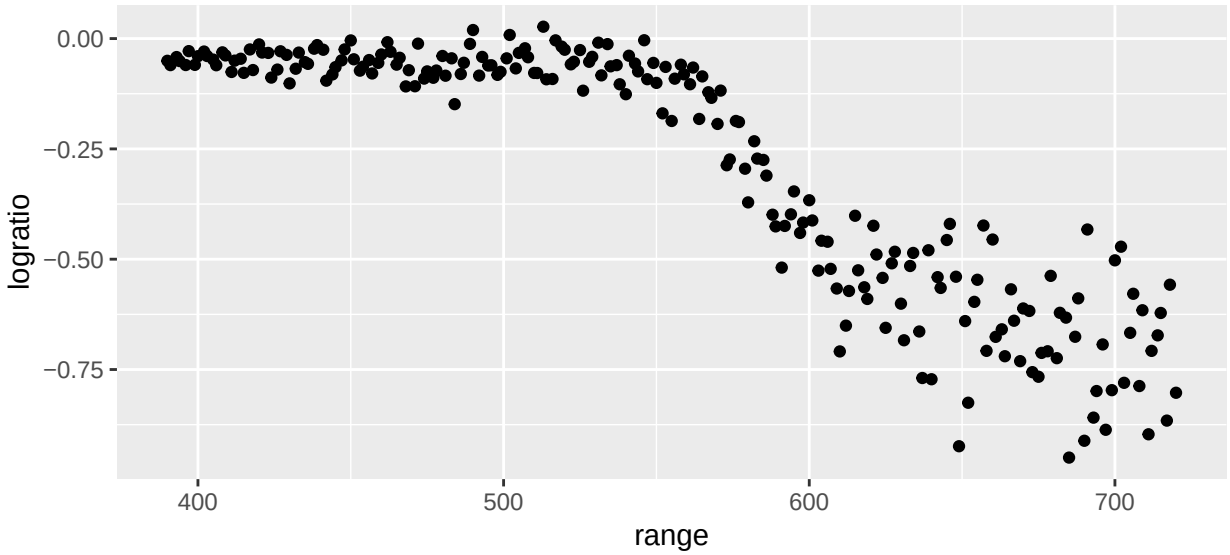
Code adapted from my solutions to Problem Set 01.

Recall that on problem set 01, I asked you to think about the bias-variance tradeoff in the context of the LIDAR data:

```
lidar <- read_table2("http://www.evanlray.com/data/all-of-nonparametric-stat/lidar.dat")
```

```
## Parsed with column specification:
## cols(
##   range = col_integer(),
##   logratio = col_double()
## )
```

```
ggplot(data = lidar, mapping = aes(x = range, y = logratio)) +
  geom_point()
```



In my solutions, I examined what might happen if we divided the data into training and test sets and calculated the test set MSE for different values of the polynomial degree. Here's code that does something like that.

```
# Read in data
cars <- read_csv("http://www.evanlray.com/data/sdm4/Cars.csv")
```

```
## Warning: Duplicated column names deduplicated: 'MPG' => 'MPG_1' [9],
## 'Weight' => 'Weight_1' [10]
```

```
## Parsed with column specification:
## cols(
##   Country = col_character(),
##   Car = col_character(),
##   MPG = col_double(),
##   Weight = col_double(),
##   `Drive Ratio` = col_double(),
##   Horsepower = col_integer(),
##   Displacement = col_integer(),
##   Cylinders = col_integer(),
##   MPG_1 = col_double(),
##   Weight_1 = col_double()
## )
```

```
# Divide into training and test sets
n <- nrow(lidar)
train_inds <- sample(n, size = 175, replace = FALSE)
test_inds <- seq_len(n)[-train_inds]
```

```r
train_data <- lidar[train_inds, ]
test_data <- lidar[test_inds, ]

# Function we will call multiple times.
# It returns a numeric value (real number, i.e. "double" in R's system of data types)
get_test_mse <- function(degree) {
  fit <- lm(logratio ~ poly(range, degree), data = train_data)
  test_preds <- predict(fit, data.frame(range = test_data$range))
  test_resids <- test_data$logratio - test_preds
  mean(test_resids^2)
}

# Using map_dbl to call get_test_mse multiple times and assemble the results.
#  * The first argument to map_dbl says what values to use for the argument to get_test_mse
#  * In this case, we will call get_test_mse 10 times with the corresponding values of degree
#  * Since we used the version of map ending in _dbl, the results are assembled into
#    a numeric vector
mse_results <- map_dbl(1:10, get_test_mse)
mse_results
```

```
##  [1] 0.016357542 0.007134309 0.007898422 0.005732242 0.006110128
##  [6] 0.006058590 0.005856546 0.005336987 0.005331829 0.004713635
```

```r
typeof(mse_results)
```

```
## [1] "double"
```

```r
# Our goal might be to find out which polynomial degree has lowest test set MSE.
min(mse_results)
```

```
## [1] 0.004713635
```

```r
which.min(mse_results)
```

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

pmap: Your function has more than 1 argument that is different for each function all.

Example: Assemble output into a numeric vector. pmap_dbl

We haven't had to do this yet for anything real, so here's a made up example. (We will be using this functionality next class!)

```
arguments <- expand.grid(
  a = c(3, 4),
  b = c(5, 6)
)
arguments
```

```
##   a b
## 1 3 5
## 2 4 5
## 3 3 6
## 4 4 6
```

```
add_a_b <- function(a, b) {
  return(a + b)
}

pmap_dbl(arguments, add_a_b)
```

```
## [1]  8  9  9 10
```