HW 04 - Grades Visualizer

Stat 133, Fall 2017, Prof. Sanchez

Due date: Sun Nov-19 (before midnight)

Introduction

In this assignment you'll be working on a data set containing raw scores of fictitious students in a hypothetical Stat 133 course. Overall, this HW will give you the opportunity to work on a relatively *small* data computing and visualization project. By *small* we don't mean simple; there's actually a fair amount of complexity that you will have to deal with.

From the user point of view, the main deliverable will be a shiny app to visualize: 1) the overall grade distribution, 2) the distribution and summary statistics of various scores, and 3) the relationships between pairs of scores.

From the developer point of vew, you will have to write a number of functions that help you process the data, and compute the required statistics. In addition, you will have to write unit tests for the programmed functions, which is an essential part of any programming task.

In summary, this project involves working around three primary aspects:

- Low level coding:
  - writing functions (and document them)
  - testing functions (runing unit tests)
- Data Analysis Cycle:
  - data preparation, and reformatting
  - data analysis and visualization
  - reporting via interactive tools
- Practice with R packages:
  - `"testthat"`
  - `"shiny"`
  - `"ggvis"`
  - optional: `"readr"`, `"dplyr"`, etc

As a starting point, you can look at a simple demo in the shiny app `ggvis-demo`:

https://github.com/ucb-stat133/stat133-fall-2017/tree/master/apps/ggvis-demo

This app should give you an example of some of the things you have to work on this assignment and how to start approaching them.

1) File Structure

After completing your assignment, the file structure of your project should look like this:

```
hw04/
   README.md
   data/
      rawdata/
         rawscores.csv
         rawscores-dictionary.md
      cleandata/
         cleanscores.csv
         cleanscores-dictionary.md
   code/
      functions.R
      tests.R
      tester-script.R
      clean-data-script.R
   output/
      test-reporter.txt
      summary-rawscores.txt
      Lab-stats.txt
      Homework-stats.txt
      Quiz-stats.txt
      Test1-stats.txt
      Test2-stats.txt
      Overall-stats.txt
   app/
      gradevis.R
```

In your `README.md` file include a code block, using your own `username`, with the command that a user can invoke on RStudio to run your shiny app (assuming they have permission to access you private repo):

```r
library(shiny)

# Run an app from a subdirectory in the repo
runGitHub("stat133-hws-fall17", "username", subdir = "hw04/app")
```

Use relative file paths; do NOT set working directories e.g.

```
# absolute paths break reproducibility!!!
# (don't do things like this)
setwd('~/Dropbox/stat133-hws-fall17/hw04')
```

2) Functions

You will have to create the following functions in an `R` script `functions.R` (to be stored inside the `code/` folder):

- `remove_missing()`
- `get_minimum()`
- `get_maximum()`
- `get_range()`
- `get_percentile10()`
- `get_percentile90()`
- `get_quartile1()`
- `get_quartile3()`
- `get_median()`
- `get_average()`
- `get_stdev()`
- `count_missing()`
- `summary_stats()`
- `print_stats()`
- `drop_lowest()`
- `rescale100()`
- `score_homework()`
- `score_quiz()`
- `score_lab()`

When writing your functions, keep in mind these considerations:

- Document all your functions including: a title, a description, input parameters (i.e. arguments), and returned value (i.e. output).
- Do NOT write lines of code that exceed a width of 80 characters. In other words, break a long line of code into several small lines.
- Do NOT write functions with more than 15 lines of code (excluding comments).
- If you have a function that exceeds 15 lines of code, then you should break it down into two or more simpler functions. This means you can add auxiliary functions to those listed above.
- All the functions of the form `get_...()` e.g. `get_minimum()`, `get_median()`, should return an error message if the provided input is not a numeric vector: e.g. something like `"non-numeric argument"`

Function `remove_missing()`

Write a function that takes a vector, and returns the input vector without missing values. You are NOT allowed to use `na.omit()` or `complete.cases()`.

```r
# example of remove_missing()
a <- c(1, 4, 7, NA, 10)
remove_missing(a)
```

```
## [1]  1  4  7 10
```

Function `get_minimum()`

Write a function that takes a numeric vector, and an optional logical `na.rm` argument, to find the minimum value. If `na.rm = TRUE`, you should call `remove_missing()` inside `get_minimum()`. You can use the function `sort()` to implement `get_minimum()` but you are NOT allowed to use `min()`.

```r
# example of get_minimum()
a <- c(1, 4, 7, NA, 10)
get_minimum(a, na.rm = TRUE)
```

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

Function `get_maximum()`

Write a function that takes a numeric vector, and an optional logical `na.rm` argument, to find the maximum value. If `na.rm = TRUE`, you should call `remove_missing()` inside `get_maximum()`. You can use the function `sort()` to implement `get_maximum()` but you are NOT allowed to use `max()`.

```r
# example of get_maximum()
a <- c(1, 4, 7, NA, 10)
get_maximum(a, na.rm = TRUE)
```

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

Function `get_range()`

Use `get_minimum()` and `get_maximum()` to write a function that takes a numeric vector, and an optional logical `na.rm` argument, to compute the overall range of the input vector. If `na.rm = TRUE`, you should call `remove_missing()` inside `get_range()`. You are NOT allowed to use `range()`.

$$range = max(x) - min(x)$$

```r
# example of get_range()
a <- c(1, 4, 7, NA, 10)
get_range(a, na.rm = TRUE)
```

```
## [1] 9
```

Function `get_percentile10()`

Write a function that takes a numeric vector, and an optional `na.rm` argument, to compute the 10th percentile of the input vector. You can use `quantile()` to create `get_percentile10()`

```r
# example of get_percentile10()
a <- c(1, 4, 7, NA, 10)
get_percentile10(a, na.rm = TRUE)
```

```
## [1] 1.9
```

Function `get_percentile90()`

Write a function that takes a numeric vector, and an optional `na.rm` argument, to compute the 90th percentile of the input vector. You can use `quantile()` to create `get_percentile90()`

```r
# example of get_percentile90()
a <- c(1, 4, 7, NA, 10)
get_percentile90(a, na.rm = TRUE)
```

```
## [1] 9.1
```

Function `get_median()`

Write a function that takes a numeric vector, and an optional logical `na.rm` argument, to compute the median of the input vector. If `na.rm = TRUE`, you should call `remove_missing()` inside `get_median()`. You can use `sort()` to implement `get_median()` but you are NOT allowed to use `median()`.

To find the median:

- Put all the numbers in numerical order.
- If there is an odd number of results, the median is the middle number.
- If there is an even number of results, the median will be the mean of the two central numbers.

```r
# example of get_median()
a <- c(1, 4, 7, NA, 10)
get_median(a, na.rm = TRUE)
```

```
## [1] 5.5
```

Function `get_average()`

Write a function that takes a numeric vector, and an optional logical `na.rm` argument, to compute the average (i.e. mean) of the input vector. If `na.rm = TRUE`, you should call `remove_missing()` inside `get_average()`. You should use a `for()` loop to compute the average. In other words, you are NOT allowed to use the functions `mean()` or `sum()`

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

```r
# example of get_average()
a <- c(1, 4, 7, NA, 10)
get_average(a, na.rm = TRUE)
```

```
## [1] 5.5
```

Function `get_stdev()`

Write a function that takes a numeric vector, and an optional logical `na.rm` argument, to compute the standard deviation of the input vector. If `na.rm = TRUE`, you should call `remove_missing()` inside `get_stdev()`. You should use a `for()` loop to compute the standard deviation, and you should call `get_average()` inside `get_stedv()`. However, you are NOT allowed to use the functions `var()`, `sd()`, or `sum()`.

$$SD = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

```r
# example of get_stdev()
a <- c(1, 4, 7, NA, 10)
get_stdev(a, na.rm = TRUE)
```

```
## [1] 3.872983
```

Function `get_quartile1()`

Write a function that takes a numeric vector, and an optional `na.rm` argument, to compute the first quartile of the input vector. You can use `quantile()` to create `get_quartile1()`

```r
# example of quartile1()
a <- c(1, 4, 7, NA, 10)
get_quartile1(a, na.rm = TRUE)
```

```
## [1] 3.25
```

Function `get_quartile3()`

Write a function that takes a numeric vector, and an optional `na.rm` argument, to compute the third quartile of the input vector. You can use `quantile()` to create `get_quartile3()`

```r
# example of get_quartile3()
a <- c(1, 4, 7, NA, 10)
get_quartile3(a, na.rm = TRUE)
```

```
## [1] 7.75
```

Function `count_missing()`

Write a function that takes a numeric vector, and calculates the number of missing values `NA`.

```r
# example of count_missing()
a <- c(1, 4, 7, NA, 10)
count_missing(a)
```

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

Function `summary_stats()`

Use the previous functions to write a function that takes a numeric vector, and returns a **list** of summary statistics (see example below):

```r
# example of summary_stats()
a <- c(1, 4, 7, NA, 10)
stats <- summary_stats(a)
stats
```

```
## $minimum
## [1] 1
##
## $percent10
## [1] 1.9
##
## $quartile1
## [1] 3.25
##
## $median
## [1] 5.5
##
```

```
## $mean
## [1] 5.5
##
## $quartile3
## [1] 7.75
##
## $percent90
## [1] 9.1
##
## $maximum
## [1] 10
##
## $range
## [1] 9
##
## $stdev
## [1] 3.872983
##
## $missing
## [1] 1
```

Function `print_stats()`

Write a function that takes a list of summary statistics, and prints the values in a *nice* format, like in the following example:

```
# example of print_stats()
print_stats(stats)
```

```
## minimum  : 1.0000
## percent10: 1.9000
## quartile1: 3.2500
## median   : 5.5000
## mean     : 5.5000
## quartile3: 7.7500
## percent90: 9.1000
## maximum  : 10.0000
## range    : 9.0000
## stdev    : 3.8730
## missing  : 1.0000
```

Notice the order of the statistics, the alignment of the colons `":"`, and displayed number of decimal digits.

Function `rescale100()`

Write a function that takes three arguments: a numeric vector `x`, a minimum `xmin`, and a maximum `xmax`, to compute a rescaled vector with a potential scale from 0 to 100:

$$z = 100 \times \frac{x - xmin}{xmax - xmin}$$

```r
# example of rescale100()
b <- c(18, 15, 16, 4, 17, 9)
rescale100(b, xmin = 0, xmax = 20)
```

```
## [1] 90 75 80 20 85 45
```

Function `drop_lowest()`

Write a function that takes a numeric vector of length n, and returns a vector of length $n-1$ by dropping the lowest value.

```r
# example of drop_lowest()
b <- c(10, 10, 8.5, 4, 7, 9)
drop_lowest(b)
```

```
## [1] 10.0 10.0  8.5  7.0  9.0
```

Function `score_homework()`

Write a function that takes a numeric vector of homework scores (of length n), and an optional logical argument `drop`, to compute a single homework value. If `drop = TRUE`, the lowest HW score must be dropped. The function should return the average of the homework scores—but are NOT allowed to use `mean()`.

```r
# example of score_homework()
hws <- c(100, 80, 30, 70, 75, 85)
score_homework(hws, drop = TRUE)
```

```
## [1] 82
```

```r
score_homework(hws, drop = FALSE)
```

```
## [1] 73.33333
```

Function `score_quiz()`

Write a function that takes a numeric vector of quiz scores (of length n), and an optional logical argument `drop`, to compute a single quiz value. If `drop = TRUE`, the lowest quiz score must be dropped. The function should return the average of the quiz scores—but are NOT allowed to use `mean()`.

```r
# example of score_quiz()
quizzes <- c(100, 80, 70, 0)
score_quiz(quizzes, drop = TRUE)
```

```
## [1] 83.33333
```

```r
score_quiz(quizzes, drop = FALSE)
```

```
## [1] 62.5
```

Function `score_lab()`

Write a function that takes a numeric value of lab attendance, and returns the lab score. The attendance value ranges between 0 and 12. Use the following table to compute the corresponding lab score:

input value	lab score
11 or 12	100
10	80
9	60
8	40
7	20
6 or less	0

```r
# example of score_lab()
score_lab(12)
```

```
## [1] 100
```

```r
score_lab(10)
```

```
## [1] 80
```

```r
score_lab(6)
```

```
## [1] 0
```

3) Unit Tests

Once you created your functions, the next stage consists of writing unit tests. Writing unit test should become second nature. And you should make it a habit ASAP. While writing tests definitely takes time, it's something that has a high return on investment (ROI): your code will tend to be more robust, less bug-prone, and (hopefully) easier to maintain.

Write your tests in an `R` script file named `tests.R`, to be saved inside the `code/` folder. To write the tests you will use functions from the package `"testthat"`.

- Create contexts for each function—via `context()`
- Include at least four expectations for each function (e.g. `expect_equal()`)
- Create a separate `R` script `tester-script.R`, inside the `code/` folder, and include the following lines of code:

```
# test script
library(testthat)

# source in functions to be tested
source('functions.R')

sink('../output/test-reporter.txt')
test_file('tests.R')
sink()
```

Here's a list of resources that you may need to read to know more about unit tests with `"testthat"`

http://r-pkgs.had.co.nz/tests.html

https://katherinemwood.github.io/post/testthat/

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5500893/

https://www.johndcook.com/blog/2013/06/12/example-of-unit-testing-r-code-with-testthat/

4) Data Set `rawscores.csv` and dictionary

The raw data is in the file `rawscores.csv`, available in the `data/` folder of the course github repository. Download the file and save it in the `data/rawdata/` folder.

The data set `rawscores.csv` contains the raw scores for a fictitious Stat 133 class. The columns refer to various types of assingments:

- homework assignments: columns `HW1` to `HW9`, each 100 pts
- lab attendance: `ATT`, number of attended labs (0 to 12)
- quiz scores:
  - `QZ1`, 12 pts
  - `QZ2`, 18 pts
  - `QZ3`, 20 pts
  - `QZ4`, 20 pts
- exam 1: `EX1`, 80 pts

- exam 2: `EX2`, 90 pts

Create a data dictionary file `rawscores-dictionary.md`, and place it in the folder `data/rawdata/`.

5) Data Preparation

To clean the raw data, create an `R` script `clean-data-script.R`, inside the `code/` folder. Remember to include a header, followed by the calls to load any required packages, as well as invoking `source()` to source in the functions in the script `functions.R`. Also, use **relative** file paths!

- read in the `CSV` file with the raw scores
- `sink()` the structure `str()` of the data frame of raw scores to a file `summary-rawscores.txt`, inside the `output/` folder. Include also the summary statistics, using `summary_stats()` and `print_stats()`, for all the columns in the data frame.
- write a `for()` loop, traversing the columns of the data frame, to replace all missing values `NA` with zero.
- use `rescale100()` to rescale `QZ1`: 0 is the minimum, and 12 is the max.
- use `rescale100()` to rescale `QZ2`: 0 is the minimum, and 18 is the max.
- use `rescale100()` to rescale `QZ3`: 0 is the minimum, and 20 is the max.
- use `rescale100()` to rescale `QZ4`: 0 is the minimum, and 20 is the max.
- use `rescale100()` to add a variable `Test1` by rescaling `EX1`: 0 is the minimum, and 80 is the max.
- use `rescale100()` to add a variable `Test2` by rescaling `EX2`: 0 is the minimum, and 90 is the max.
- add a variable `Homework` to the data frame of scores; this variable should contain the overall homework score obtained by dropping the lowest HW, and then averaging the remaining scores.
- add a variable `Quiz` to the data frame of scores; this variable should contain the overall quiz score obtained by dropping the lowest quiz, and then averaging the remaining scores.
- add a variable `Overall` to the data frame of scores; this variable should be calculated using the following grading structure:
  - 10% Lab score
  - 30% Homework score (drop lowest HW)
  - 15% Quiz score (drop lowest quiz)
  - 20% Test 1 score
  - 25% Test 2 score

  - (`Overall` must be already in a scale 0 to 100)

- calculate a variable `Grade` (and add it to the data frame of scores), that contains the letter grade based on the following cutting points. A bracket `[` means inclusion, and a parenthesis `)` means exclusion. For instance, consider the letter grade: `B = [82% - 86%)`, this means that a *B* corresponds to an overall score greater than or equal to 82 but less than 86.

Letter	Overall Score
`F`	[0 - 50)
`D`	[50 - 60)
`C-`	[60 - 70)
`C`	[70 - 77.5)
`C+`	[77.5 - 79.5)
`B-`	[79.5 - 82)
`B`	[82 - 86)
`B+`	[86 - 88)
`A-`	[88 - 90)
`A`	[90 - 95)
`A+`	[95 - 100]

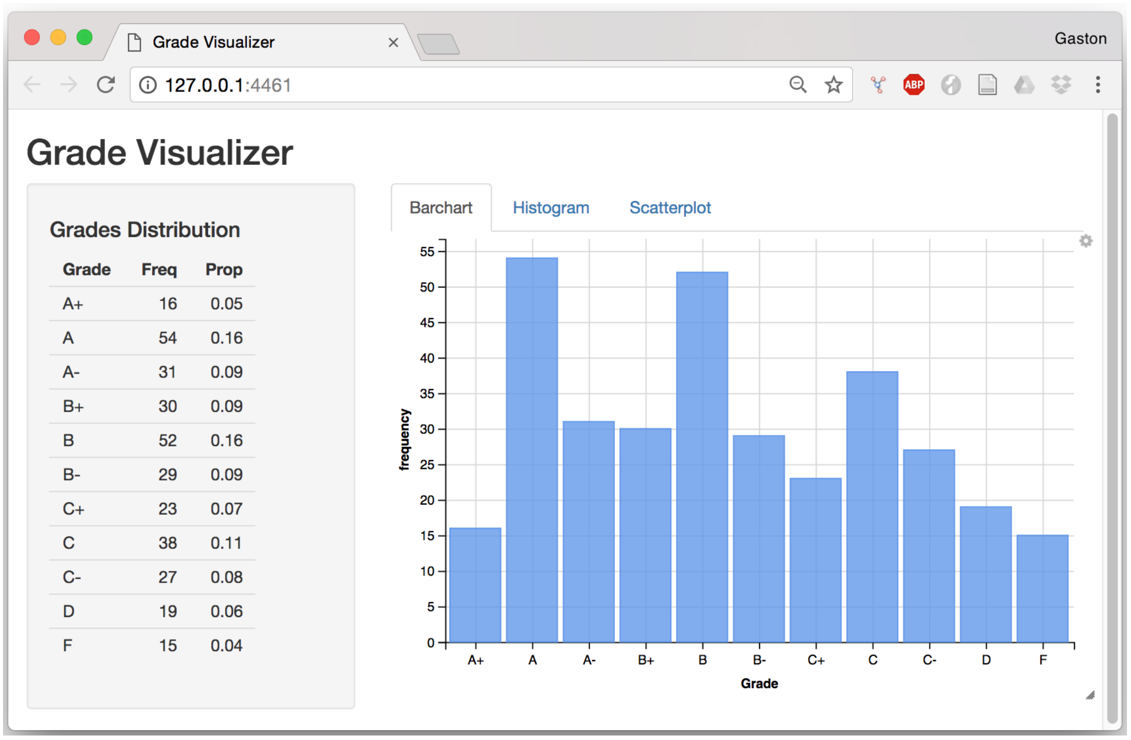
- write a for loop in which you use your functions `summary_stats()` and `print_stats()` to export, via `sink()`, the summary statistics for `Lab`, `Homework`, `Quiz`, `Test1`, `Test2`, and `Overall`. The summary statistics should be `sink()`ed into separated text files, inside the `output/` folder:
  - `Lab-stats.txt`
  - `Homework-stats.txt`
  - `Quiz-stats.txt`
  - `Test1-stats.txt`
  - `Test2-stats.txt`
  - `Overall-stats.txt`
- `sink()` the structure `str()` of the data frame of clean scores to a file `summary-cleanscores.txt`, inside the `output/` folder.
- Finally, export the clean data frame of scores to a CSV file `cleanscores.csv` inside the folder `data/cleandata/`. This data set should contain 334 rows, and 23 columns.

6) Shiny App

After the data preparation, you will have to create a shiny app that allows you to visualize the grades from different perspectives.

Your shiny app should use three tabs: barchart, histogram, scatterplot; in turn, each bar has conditional panels which allows to have different sidebar panels.

The first tab will show the table and barchart with the letter grades:

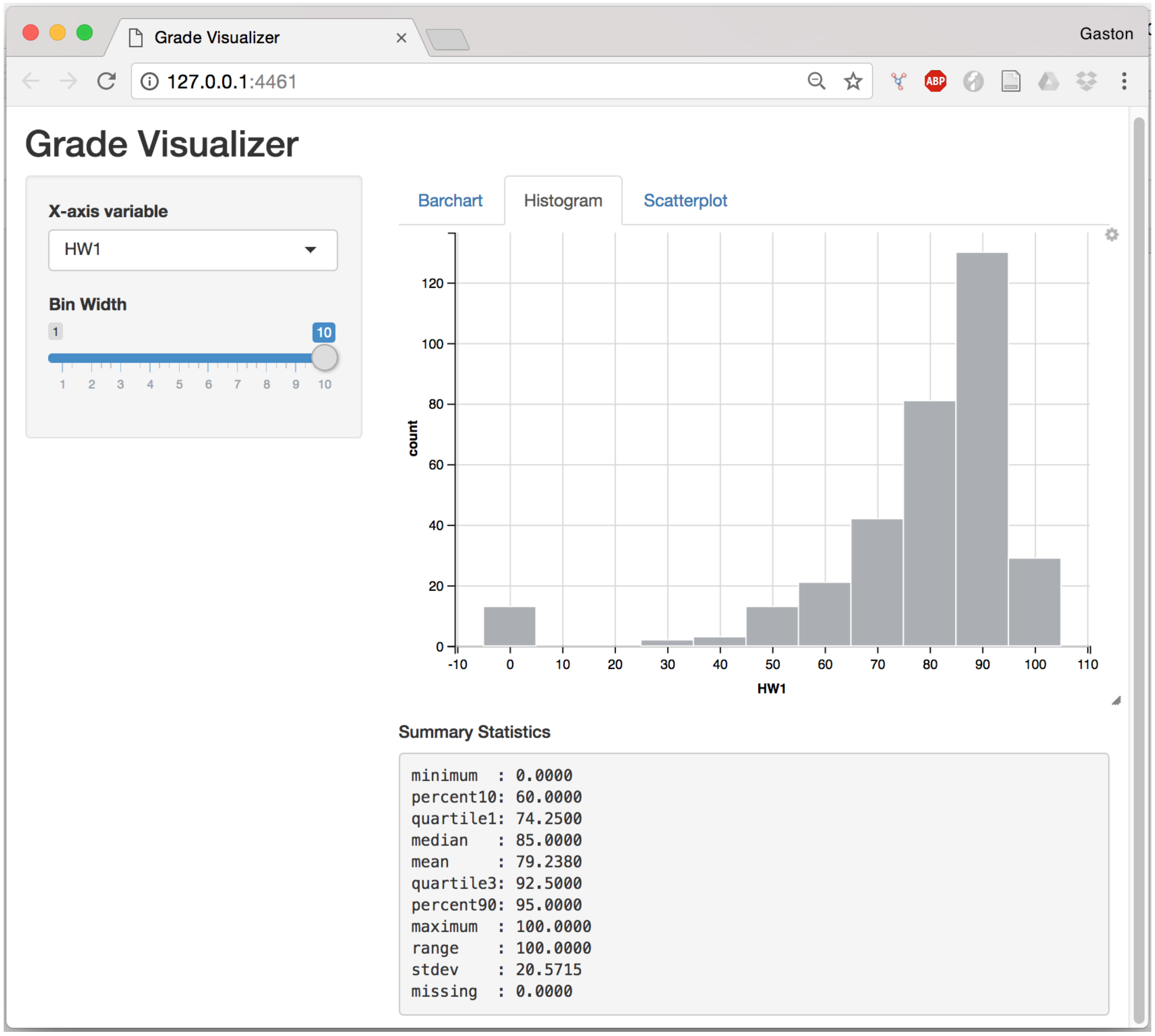


Grade Visualizer

Grades Distribution

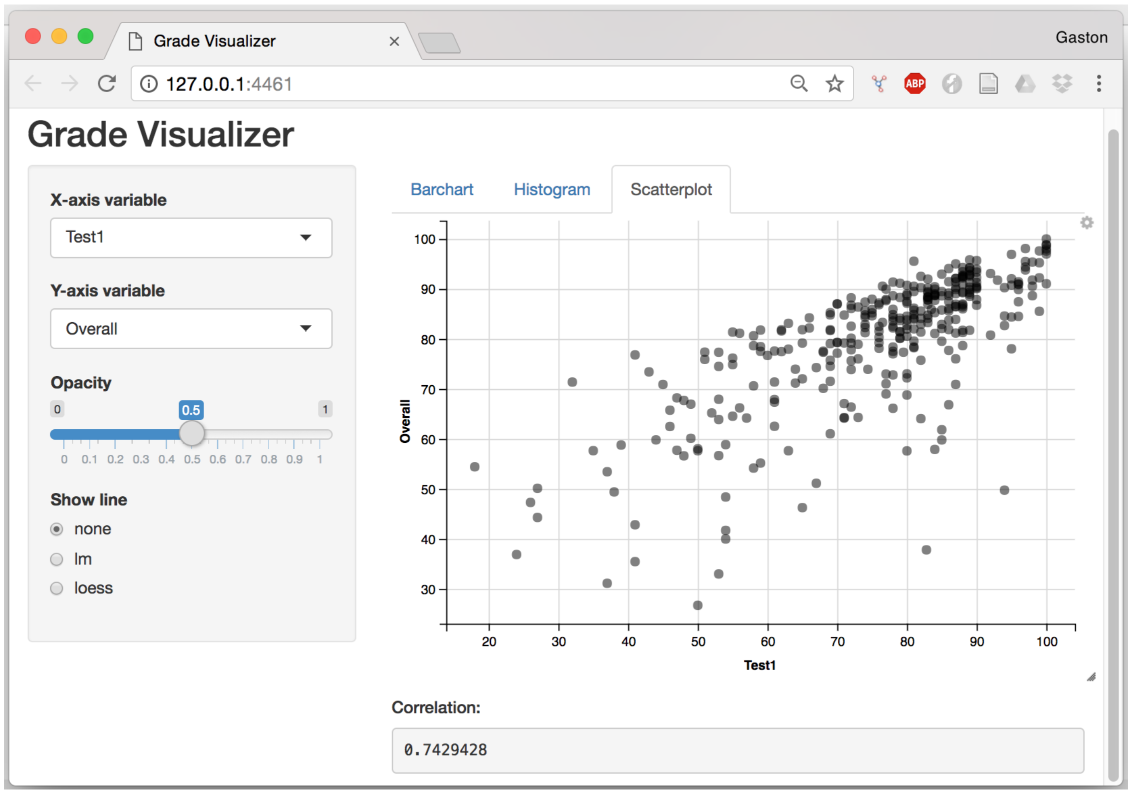
Grade	Freq	Prop
A+	16	0.05
A	54	0.16
A-	31	0.09
B+	30	0.09
B	52	0.16
B-	29	0.09
C+	23	0.07
C	38	0.11
C-	27	0.08
D	19	0.06
F	15	0.04

The second tab will show the histogram and summary statistics for a selected variable:



```
minimum  : 0.0000
percent10: 60.0000
quartile1: 74.2500
median   : 85.0000
mean     : 79.2380
quartile3: 92.5000
percent90: 95.0000
maximum  : 100.0000
range    : 100.0000
stdev    : 20.5715
missing  : 0.0000
```

The third tab will show scatterplots between a pair of selected variables, the correlation coefficient, and optionally the regression line or the loess line:



Grade Visualizer

X-axis variable: Test1

Y-axis variable: Overall

Opacity: 0.5

Show line: [✓ none] lm loess

Correlation:

```
0.7429428
```

7) Comments and Reflections

Reflect on what was hard/easy, problems you solved, helpful tutorials you read, etc.

- Was this your first time writing unit tests?
- On a scale from 0 to 10, how confusing you found the logic of `testthat` tests? (0 not at all, 10 very confusing)
- Was this your first time working with `ggvis`?
- On a scale from 0 to 10, how confusing you found the syntax of `ggvis`? (0 not at all, 10 very confusing)
- Was this your first time working with conditional panels in `shiny`?
- On a scale from 0 to 10, how challenging you found to work with the conditional panels? (0 not at all, 10 very challenging)
- So far we've exposed you to three graphing paradigms in R: base plots, ggplot, and now ggvis. Which do you like the most and why?
- Did anyone help you completing the assignment? If so, who?
- How much time did it take to complete this HW?
- What was the most time consuming part?