Lab 3

Stat 131A

September 13 and 15, 2021

Welcome to the Lab 3! In this lab, you will:

1) Take a closer look on kernel density estimation
2) Implement a permutation test
3) Implement a T-test
4) Illustrate the difference between a T-distribution and the standard normal distribution
5) Illustrate how functions work in `R`

We will continue to use the rent price dataset from the Lab 2.

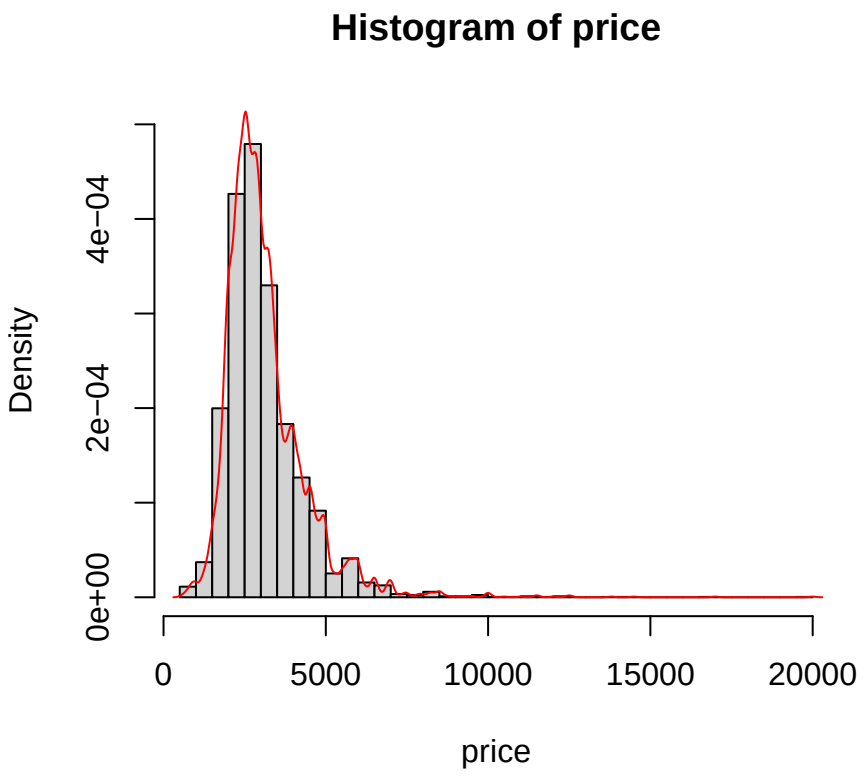
Read in data:

```
craigslist <- read.csv("craigslist.csv", header = TRUE)
```

Kernel density estimation

In Lab 2, we plotted a kernel density curve of the monthly rents as shown below. For this exercise, experiment with the `bw` argument in the density function, which stands for the bandwidth (width of the the moving window) of the kernel density function. For example, start with `bw = 1`, and scale up by a multiple of 10, until `bw` is greater than n, where n is the number of observations. Observe how the resulting curves change.

```
price <- craigslist$price
d <- density(na.omit(price), bw = 100)
hist(price,
     freq = FALSE,
     ylim = c(0, 0.0005),
     breaks = 50)
lines(d, col = "red")
```



We can recreate the histogram and kernel density curve using `tidy`-style code. The package `tidyverse` includes packages `ggplot2`, `purrr`, `tibble`, `dplyr`, `tidyr`, `stringr`, `readr`, and `forcats`. The code below relies only on `ggplot2` and `dplyr`. Loading `tidyverse` may be more convenient than loading several `tidy` packages individually.

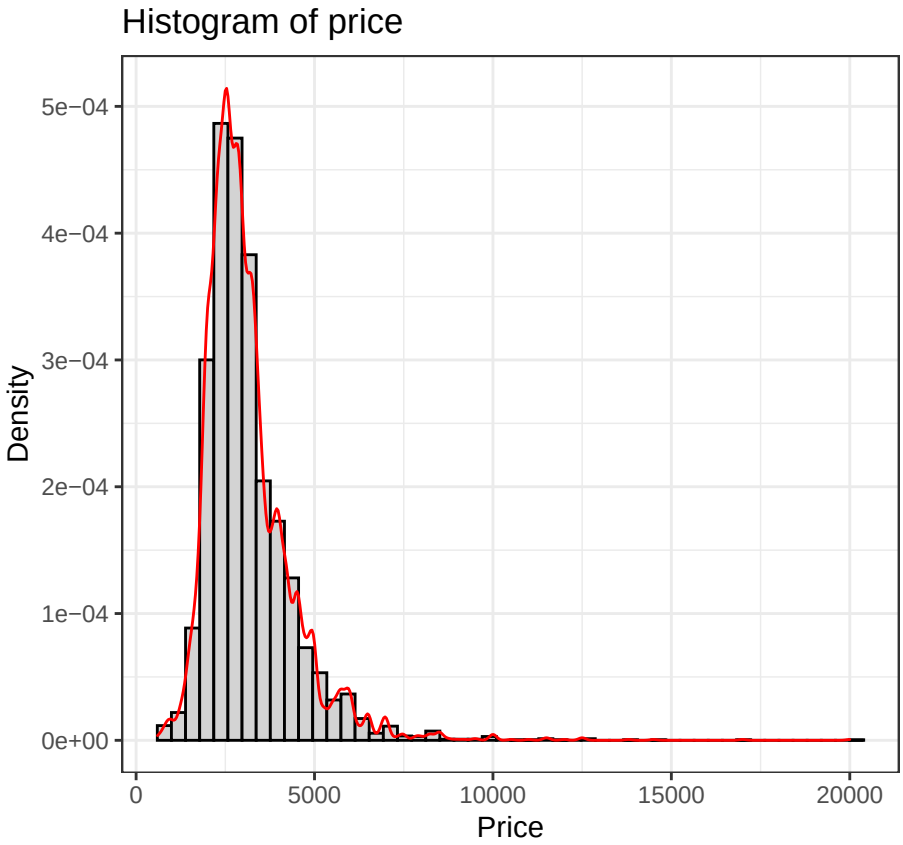
```
# install.packages("tidyverse")
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.4     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   2.0.1     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
craigslist %>% ggplot(aes(x = price, y = ..density..)) +
  geom_histogram(bins = 50, fill = "lightgrey", color = "black") +
  geom_density(bw = 100, color = "red") +
  labs(x = "Price", y = "Density") +
  ggtitle("Histogram of price") +
  theme_bw()
```



Exercise 1.

(a) What value of bandwidth (approximately) is used as default by the density function on monthly rents? (In other words, what value of the `bw` argument results in the same kernel density estimation as not specifying the `bw` argument?)
Bonus: what is the actual default bandwidth?

Using the command bw.nrd0(na.omit(price)) to get the default bandwidth. It's 147.9897 in this case.

(b) How does changing the bandwidth affect the density estimation curve? Briefly explain your intuition on why that is the case.
Hint: consider the weighted kernel function in textbook 2.5.2.2. The bandwidth corresponds to the w in the function.

If we increase the bandwidth, the density estimation curve will become smoother. Otherwise, if we narrow the bandwidth, the density estimation curve will become rougher. That's because increasing the bandwidth broadens the width of the moving window, bringing more neighboring samples into density estimation and making the density estimation curve less sensitive to specific values.

Replicate

To repeat things in `R`, there is a more convenient and efficient way than using a `for` loop. The function `replicate` is designed for evaluating an expression repeatedly. For example, to get a vector of length 50 where each element is generated independently from the sum of 5 normally distributed samples:

```
set.seed(20172828)
example1 <- replicate(50, sum(rnorm(5)))
```

The equivalence of the above code using for loops will be:

```r
# Create a vector of length 0 to store the results
example2 <- c()
# Calculate for 50 times
for (i in 1:50) {
  # Combine the calculated sum of normal samples to the end of `results` vector
  example2 <- c(example2, sum(rnorm(5)))
}
```

The first argument of `replicate` function is the number of replications. And the second argument is an expression which will be evaluated repeatedly, which usually involves random sampling and simulation. (Otherwise, you would get a vector of one identical number. That's useless!) Sometimes, the things you want to replicate cannot finish in one line; you may need to use the big curly bracket such as the following. Elements in the `example3` vector would be the value of the last expression in the bracket. (Since we set the same seed for `example1` and `example3`, you may want to compare the value of them to see what happened.)

```r
set.seed(20172828)
example3 <- replicate(50, {
  a <- rnorm(5)
  b <- 5
  sum(a) + b    # return the value
})
```

Permutation test

Exercise 2

Here, we will perform a permutation test to compare the average one-bedroom apartment rent price in Berkeley and Palo Alto.

(a) Subset the dataset to only consider one-bedroom apartments in Berkeley and Palo Alto.

```r
# Insert code here save the data frame of one bedroom postings
# in Berkeley and Palo Alto as `craigslist.subset`
craigslist.subset <- subset(craigslist, location %in% c("berkeley", "palo alto") & brs == 1)
# View(craigslist.subset)
```

(b) Calculate the number of postings for Berkeley in the data frame `subset`.

```r
# Insert code here save the number of postings for Berkeley as
# `no.berkeley`
no.berkeley <- sum(craigslist.subset$location == "berkeley")
```

(c) Calculate the observed statistic, i.e., the absolute difference between the mean of Berkeley and mean of Palo Alto one bedroom rent price.

```r
# Insert code here save the observed statistics as
# `stat.obs`
stat.obs <- abs(mean(craigslist.subset[craigslist.subset$location == "berkeley", "price"])-
   mean(craigslist.subset[craigslist.subset$location == "palo alto", "price"]))
```

(d) Calculate the permuted statistics (absolute mean difference), repeat 1000 times.
Hint: use `sample` function to sample from a group of observations. You can use either use a `for` loop or the `replicate` function introduced above. However, it is a good practice using `replicate`.

```
# leave seed set as is
set.seed(20172828)
# insert code to save the simulated statistics as
# `stat.bootstrap`
stat.bootstrap <- replicate(1000, {
  rows = sample(1:nrow(craigslist.subset), no.berkeley, replace = F)
  abs(mean(craigslist.subset[rows, "price"]) - mean(craigslist.subset[-rows, "price"]))
})
```

(e) Calculate the p-value of our observed statistic using our approximation of the sampling distribution from part (d). We are testing if there is any difference between the two means. Our null hypothesis is that there is no difference between the two means. Our alternative hypothesis is that there is a difference.

```
# insert code here save the observed statistics as
# `p.value`
p.value <- mean(stat.bootstrap >= stat.obs)
p.value
```

```
## [1] 0.002
```

T-test

With the function `t.test`, implementing T-tests is easy in `R`. We will illustrate using a simulated example:

```
# Generate 100 samples from N(0, 1)
group1 <- rnorm(100, mean = 0, sd = 1)
# Generate 100 samples from N(0.2, 1)
group2 <- rnorm(100, mean = 0.2, sd = 1)
# perform t test
t.test(group1, group2)
```

```
##
##  Welch Two Sample t-test
##
## data:  group1 and group2
## t = -0.38556, df = 198, p-value = 0.7002
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.3497069  0.2353236
## sample estimates:
##     mean of x     mean of y
## -3.759364e-06  5.718789e-02
```

`t.test` usually print the results. To obtain the value of the t statistics or p-value, it is generally not wise or convenient to copy and paste from the printed output. Imagine you need to do 1000 T-test simultaneously, it is impossible to copy and paste every time. We need to figure out the output of `t.test`:

```
ttest.result <- t.test(group1, group2)
```

If you check the `ttest.result` object in your Environment window, you will find that it is a list of 9. And each element in the list stores some information.

```
names(ttest.result)
```

```
##  [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
##  [6] "null.value"  "stderr"      "alternative" "method"      "data.name"
```

To get the p-value, you can use either dollar sign or double square bracket, which are two ways usually used to extract elements from lists.

```
ttest.result$p.value    # with dollar sign, you can do tab completion
```

```
## [1] 0.7002346
```

```
ttest.result[['p.value']]
```

```
## [1] 0.7002346
```

Exercise 3

Now, apply T-test to compare the mean rent of one bedroom listings in Palo Alto and Berkeley.

```
# insert code here and save the t-statistic as
# `rent.1b.tstat`
paloalto <- craigslist[craigslist$location == "palo alto" & craigslist$brs == 1, "price"]
berkeley <- craigslist[craigslist$location == "berkeley" & craigslist$brs == 1, "price"]
ttest <- t.test(paloalto, berkeley)
rent.1b.tstat <- ttest$statistic
rent.1b.tstat
```

```
##        t
## 3.163719
```

```
# insert code here and save the p value as
# `rent.1b.pvalue`
rent.1b.pvalue <- ttest$p.value
rent.1b.pvalue
```

```
## [1] 0.001676094
```

T-distribution versus the normal distribution

The T-distribution has a mean of 0 and a variance of $n/(n-2)$. It is parameterized by just one parameter, the degrees of freedom. The degrees of freedom for practical purposes is equal to the sample size minus 1 for a one sample T-test. For a two sample T-test, as done above, the degrees of freedom is calculated through a formula (that you don't need to know).

The T-distribution is similar in shape to the standard normal distribution, but the relative difference in the tail regions of the distributions is significant. This implies that for hypothesis tests, the p-value we find will be quite different if we use the wrong distribution as our sampling distribution for the test statistic.

The following code finds the tail probability for a T-score (or t-statistic / t-value) of -1, with 30 degrees of freedom.

```
pt(-1, df = 30)
```

```
## [1] 0.1626543
```

We can see that this different from that of a Z-score (standard normal distribution) of -1.

```
pnorm(-1)
```

```
## [1] 0.1586553
```

The difference is larger when we change the degrees of freedom to 10.

```
pt(-1, df = 10)
```

```
## [1] 0.1704466
```

Exercise 4

(a) Find the tail probability of a t-score of -2 with 30 degrees of freedom. Divide this by the tail probability of a z-score of -2.

```
# Insert code here
# Save final answer (ratio) as `e3a`
e3a = pt(-2, df=30)/pnorm(-2)
e3a
```

```
## [1] 1.200543
```

(b) Repeat part (a) but for 10 degrees of freedom.

```
# Insert code here
# Save final answer (ratio) as `e3b`
e3b = pt(-2, df=10)/pnorm(-2)
e3b
```

```
## [1] 1.612914
```

It is common to run a hypothesis test with a significance level of 5 percent, or 0.05. This corresponds to rejecting the null hypothesis if we are more than 2 standard errors from the mean, or a z-score of less than -2 or more than 2. However, if we do not know the true standard error, and we use the sample standard deviation to estimate it, then we must use the T-distribution. You can see from above that in this case our true p-value is much greater than we would otherwise think it is under the standard normal distribution. This is the effect of the uncertainty that comes from not knowing the true population standard deviation.

apply, sapply, and function

Previously, we introduced `replicate` to repeat a expression several times. Now what if we want to run a function several times, but change the argument every time we run it? For example, to calculate the square root of integers from 1 to 100. Though `for` loops would certainly work, there is a much faster and simpler way in `R`. Consider the two functions `apply` and `sapply`.

`apply` traverses row- or column-wise and applies a function to each row (or column). It is usually used on matrix and data frames. Depending on the function, it returns a vector, array, or list of values.

`sapply` traverses every element in an array or a list and applies a function on each element.

Let us look at the problem of calculating the square root of integers plus the integer value itself from 1 to 100. To implement with for loops:

```
sqroots = c()
for (i in 1:100) {
  sqroots = c(sqroots, i + sqrt(i))
}
```

And it is equivalent to the following with `sapply`:

```
sqroots <- sapply(1:100, function(x) {
  x + sqrt(x)
})
```

`sapply` usually results in a shorter run time and easier code implementation.

Now we create a matrix with 100 rows and 10 columns with each entries being a random number from $N(0, 1)$.

```
mat <- matrix(rnorm(1000), 100)
```

To obtain the maximum number of each row with `apply`:

```
row.max <- apply(mat, 1, max)
```

To obtain the sum of the third the the fifth element each row with `apply`:

```
row.sum35 <- apply(mat, 1, function(x) x[3] + x[5])
```

To obtain the maximum number of each column with `apply`:

```
col.max <- apply(mat, 2, max)
```

As you can see above, a function is different from an expression in that a function can take inputs and give an output specific to that input. Functions can also be given names, and saved in the environment. What a function executes can be wrapped in curly braces if there are multiple lines. In the function declaration, the parameters of the function must be named and listed. In the body of the function, those named parameters can be called on and operated upon. The value returned by a function does not have to be coded explicitly. They return the last output if `return()` is not invoked.

```
lab131 <- function(x, yy){
  x-yy
  yy-x}

lab131(5,3)
```

```
## [1] -2
```

```
lab131 <- function(x, yy){
  return(x-yy)
  yy-x}
```

```
lab131(5,3)
```

```
## [1] 2
```

```
lab131 <- function(x, yy){

  }

lab131(5,3)
```

```
## NULL
```

Exercise 5

Write a function called 'stat131' with parameters called `st` and `lb` (no quotes) that adds the square root of `st` to `lb` and returns the resulting sum.

```
# Insert answer here
stat131 <- function(x, y) { return(sqrt(x) + y)}
```

(3a) 3.1637194

(3b) 0.0016761

(4a) 1.2005435

(4b) 1.6129145

(5) 16