Lab 10 - Iteration and Simulation - Solution

For class Tuesday, April 19

Lab Purpose

This lab is designed to introduce you to some key concepts in iteration and simulation, with some example simulations. Simulation is a valuable statistical tool. You may find that it is easier to demonstrate a result via simulation than to solve analytically. The power of modern computation is a major boost for the field in this way. If you continue your studies in statistics, you will see simulation used in a variety of places, and could even use it yourself in a Stat 495 paper or thesis.

Many simulations involve generating random numbers, so we'll use some distribution functions in base R in the lab. We'll also look at the multivariate normal distribution, model summarizing, and timing simulations, which will use the following packages:

- MASS (do not load - we will call directly due to conflicts with tidyverse)
- mvtnorm
- plotly
- broom (you may have seen in Stat 135/230, if taken)
- microbenchmark (bench was similar in the prep/textbook)

Building blocks of simulation

The two building blocks of simulation are iteration and random data generation, whether based on a data set or not. In Chapter 7, for iteration, you learned about `for()` loops and the concept of vectorization. The next couple sections will walk through some related skills, and then you'll have a chance to work on your simulation skills, with increasing challenge as the lab progresses!

Intro 1 - Iteration

We've seen some `for` loops in this class (webscraping of Emily Dickinson poems; creation of elbow plot in k-means clustering), but we haven't spent much time implementing them yet. We will get that time today!

Remember your book reminds you that if you can avoid `for` loops, through the use of vectorized functions, do that instead. That said, `for` loops are extremely useful, and a good place to start if you aren't sure about how else to get started but you know you need to iterate.

General format

The most basic format of a `for()` loop is shown below in pseudocode:

```
for (i in sequence){
  do this
}
```

For instance, we can iterate through the sequence of numbers 1 to 10 and print 10 times each number with the following code.

```
for (i in 1:10){
  print(10*i)
}
```

```
[1] 10
[1] 20
[1] 30
[1] 40
[1] 50
[1] 60
[1] 70
[1] 80
[1] 90
[1] 100
```

Of course, the vectorized version of this is much shorter:

```
10*1:10
```

```
 [1]  10  20  30  40  50  60  70  80  90 100
```

Specifying values to iterate over

In the code above, `1:10` quickly creates a vector of the numbers 1 to 10. The function `seq()` is another way to quickly create sequence vectors. For example, if we wanted to iterate through the sequence {10, 15, 20, 25, 30}, we could use the following code (see the help documentation for more details and related functions):

```
seq(10, 30, by = 5)
```

```
[1] 10 15 20 25 30
```

In R, we can iterate over any vector in a `for()` loop, even character vectors! A simple example is shown below.

```
# specify vector to iterate over
mycolors <- c("turquoise", "burgundy", "navy blue")

# Print sentence, iterating over colors in vector
for (j in mycolors){
  cat(paste0("My favorite color is ", j, "!\n"))
}
```

```
My favorite color is turquoise!
My favorite color is burgundy!
My favorite color is navy blue!
```

Storing results

Often it is useful to save results in a vector or dataframe. When doing so, it is best practice (in terms of computation time and memory) to initialize an empty vector or data frame first that gets filled in as the `for()` loop iterates. For example:

Note: wrapping a saved object in parentheses tells R to print the output of that saved object so you don't have to re-type the name of the object to view it.

```
# Initialize empty vector and view it
(vec <- rep(NA, 10))
```

```
 [1] NA NA NA NA NA NA NA NA NA NA
```

```r
# Fill vector as you iterate
for (i in 1:10){
  vec[i] <- 10*i
}

# What's in vec now?
vec
```

```
 [1]  10  20  30  40  50  60  70  80  90 100
```

```r
# Shorter vectorized version
(vec <- 10*1:10)
```

```
 [1]  10  20  30  40  50  60  70  80  90 100
```

`for()` loops vs. vectorization

`For` loops require one iteration of a procedure to be done before moving on to the next iteration, even if each iteration can operate independently (e.g., the procedure in the next iteration does not depend on the results of any previous iteration). *Vectorization*, on the other hand, allows independent iterations to run in parallel (at the same time).

Depending on the procedure, `for` loops can be relatively slow compared to vectorization, although advancements in computer hardware in recent years make this much less of a problem than it used to be. Remember there is a lot more detail in Chapter 7: Iteration and how it can be used in simulation in Chapter 13: Simulation.

Intro 2 - Random number generation

There are a number of useful functions in base R to help with generating random numbers from specific distributions, including but not limited to the distributions shared below.

Univariate distributions

Note: The `n` argument in the functions below is always the number of values we want to generate, and the remaining arguments are the parameters of the corresponding distribution that uniquely specify its shape. These parameters vary by distribution and have different impacts on the shapes. If you take Probability, you will encounter these distributions and learn more about these parameters. I've tried to describe some general characteristics about them below, but there is much more to learn here.

Continuous distributions

- Normal: `rnorm(n = , mean = , sd = )` - quintessential bell-shape
- Uniform: `runif(n = , min = , max = )` - flat, constant on a range
- Exponential: `rexp(n = , rate = )` - right-skewed, special case of a Gamma, starts at 0

Discrete distributions

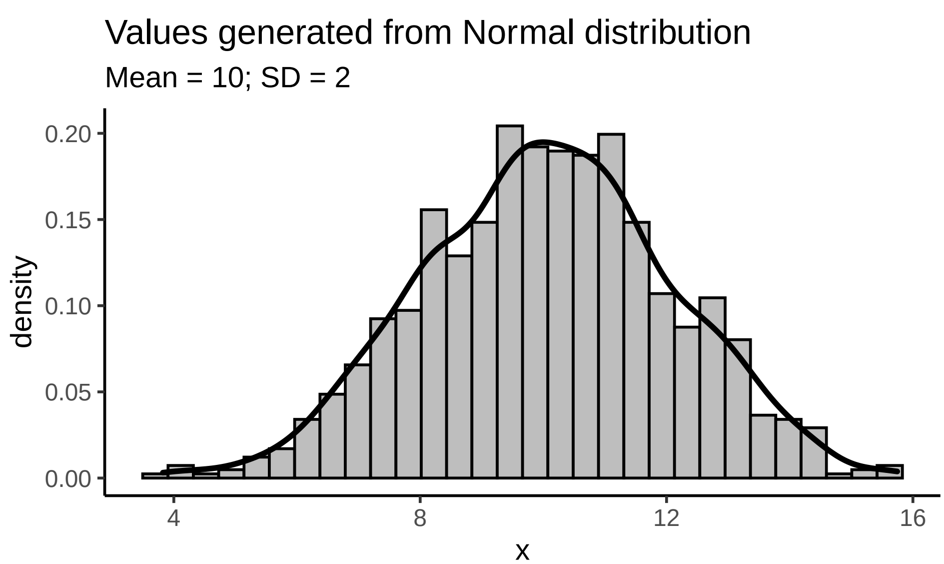
- Poisson: `rpois(n = , lambda = )` - counts number of `occurrences` in a time frame
- Binomial: `rbinom(n = , size = , p = )` - counts number of successes out of a fixed number of trials (size) (If you have seen the binomial before, the size parameter is usually referred to as n.)
- Bernoulli: `rbinom(n = , size = 1, p = )` - takes values of either 0 or 1, for one trial.

When we are generating data randomly, it is important to set the seed of R's random number generator so that we can reproduce the same results later. Generally, we only need to set the seed once before we run an entire simulation, no matter how big. *When we are troubleshooting or coding interactively, it is convenient to set a seed within each code chunk where we are coding random processes.* If you do this, it's fine to leave it (you don't need to go back to remove seeds - just leave them).

The code below generates random numbers from a Normal distribution and then creates a histogram of the generated values.

Note: Remember that ggplot requires a dataframe. Thus, the vector is turned into a dataframe for the sole purpose of plotting with `ggplot()`, and the `..density..` variable built into `ggplot()` is used to plot the density of the values instead of the frequency so we can overlay a density curve onto the histogram.

```
set.seed(231)

# Vector of values from Normal distribution
norm_vec <- rnorm(n = 1000, mean = 10, sd = 2)

# Plot of generated values (vector must be dataframe for use with ggplot)
ggplot(data.frame(norm_vec), aes(x = norm_vec, y = ..density..)) +
  geom_histogram(color = "black", fill = "grey") +
  geom_density(size = 1) +
  labs(x = "x",
       title = "Values generated from Normal distribution",
       subtitle = "Mean = 10; SD = 2")
```



Multivariate Normal distribution

The distributions above are all *univariate* (single variable) distributions. However, many scenarios involve multiple, jointly distributed variables. The most common distribution used in such cases is the *multivariate Normal* distribution. In this distribution, variables are jointly normally distributed, and it turns out, their marginal distributions are normal as well. The distribution has many nice properties, and so, it is a common assumption for many statistical procedures that the data follows a multivariate normal distribution. The variables can be correlated. In a 2-D setting, data points generated from this model would create some sort of ellipse, and in 3-D, an ellipsoid object.

We can generate data from the multivariate Normal distribution using `MASS::mvrnorm(n = , mu = , Sigma = )`, where `mu` is the mean vector, `Sigma` is the covariance matrix, and the dimensions of the mean vector and covariance matrix match the number of jointly distributed variables. If you are more familiar with correlation, and correlation matrices, the correlation matrix is a scaled version of the covariance matrix.

For example, the code below generates data from a bivariate Normal distribution, meaning there are 2 jointly distributed variables. We will plot the data in the next part of the lab, but for now run the code below to see how the parameters are specified and how the output changes if you go from `n = 1` to `n = 5` generated observations (you should see that each of the 2 jointly-distributed variables has its own column of generated values!).

```
set.seed(231)

# Specify 2-dimensional mean vector
# first variable mean is 0, second variable mean is 2
# if you wanted both to be 0, could do rep(0, 2)
(mean_vector <- c(0, 2))
```

```
[1] 0 2
```

```
# Specify 2x2 identity matrix as the covariance matrix
# This means the variables are uncorrelated and each has variance of 1
(covariance_matrix <- diag(2))
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

```
## Alternative equivalent specification of the 2x2 covariance matrix:
(covariance_matrix <- matrix(c(1, 0, 0, 1), nrow = 2))
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

```
# Generate a single observation from bivariate Normal dist
MASS::mvrnorm(n = 1, mu = mean_vector, Sigma = covariance_matrix)
```

```
[1] 2.311664 1.466898
```

```
# Generate 5 observations from bivariate Normal dist
MASS::mvrnorm(n = 5, mu = mean_vector, Sigma = covariance_matrix) %>%
  data.frame()
```

```
           X1       X2
1  0.87782986 1.045802
2 -0.56368979 2.262516
3  0.03432728 1.526643
4  0.22631292 2.217397
5 -1.38657787 2.062922
```

A note on number of iterations (or simulations) vs. sample size

We typically refer to n as the sample size in statistics. In simulations, we are usually conducting repeated sampling of samples of the same size. We must be careful, then, to distinguish between the number of times we want to simulate or iterate through a process (n_{sim}, the number of samples we are drawing, or simulation size) and the number of observations in each simulated dataset (n_{obs}, the sample size). The number of simulations may be referred to as *replications* or *reps* to help distinguish this.

Intro - 3 - Start small, then build up!

Once we have some familiarity with the building blocks of simulations, it's important to know how to build them from scratch. Regardless of which method we choose to iterate or how we generate data, the key to setting up simulations is to start small and very slowly build up before scaling to the final desired size. In other words, you want to make sure things work for a single iteration, then maybe 5 iterations, 20, 50, and so on, checking and adjusting for errors along the way.

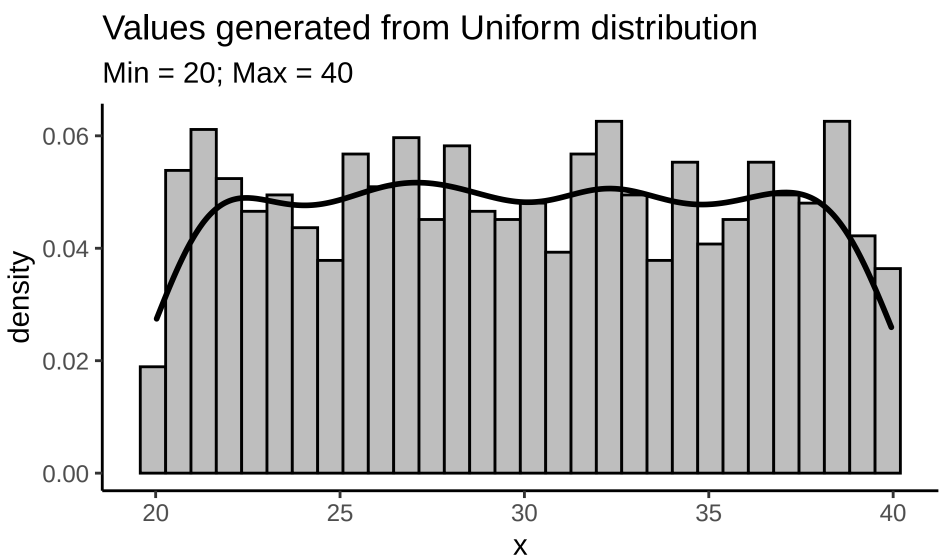
1 - Randomly generating data

part a - Univariate distributions. The code above got you started with generating random numbers from a distribution and creating a plot of the distribution. The remaining distributions are provided below. Play around with each distribution, changing the values of the parameters to see how they affect the shape, center, and spread of the distribution.

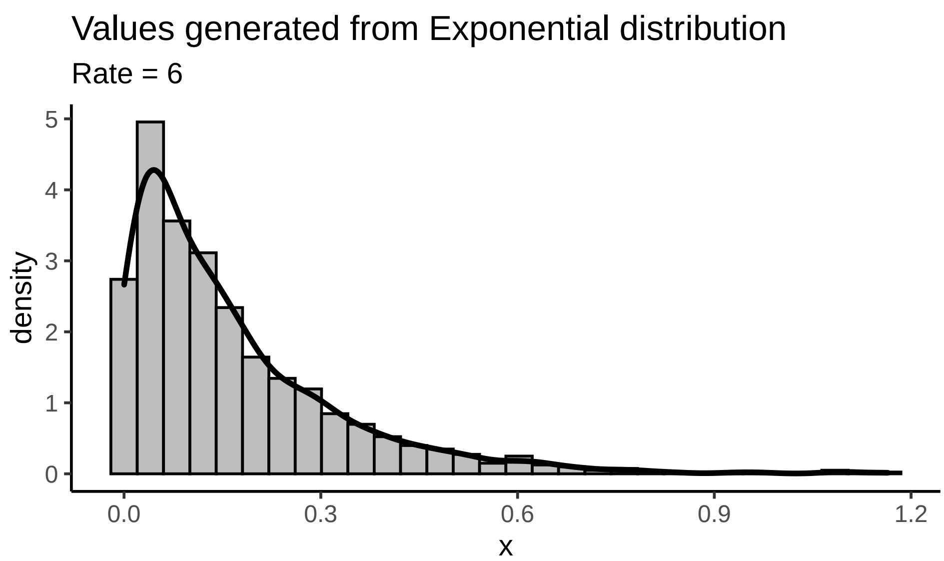
Solution:

Answers will vary. Play around with the values. Below are different values than the original code.

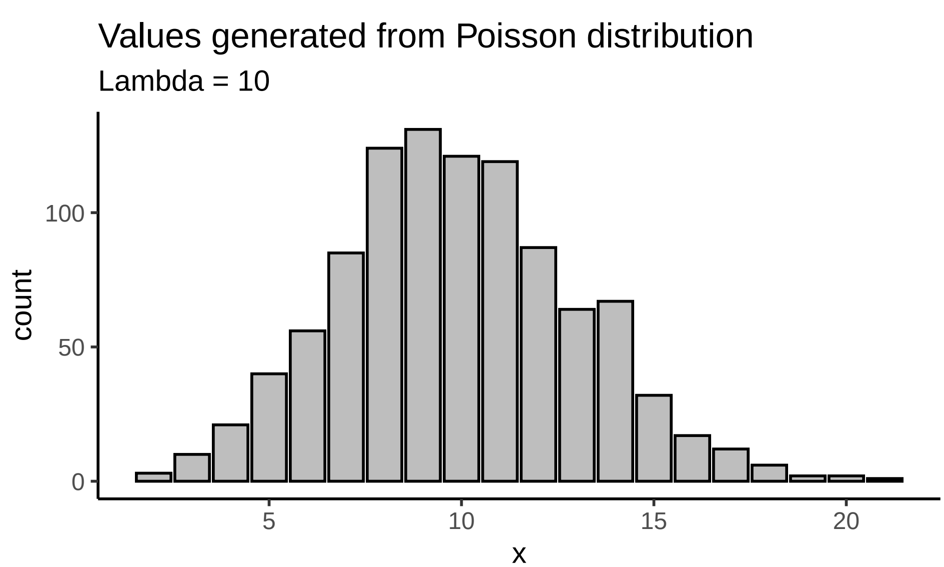
```
set.seed(8)

# Vector of values from Uniform distribution
unif_vec <- runif(n = 1000, min = 20, max = 40)

# Plot of generated values (vector must be dataframe for use with ggplot)
ggplot(data.frame(unif_vec), aes(x = unif_vec, y = ..density..)) +
  geom_histogram(color = "black", fill = "grey") +
  geom_density(size = 1) +
  labs(x = "x",
       title = "Values generated from Uniform distribution",
       subtitle = "Min = 20; Max = 40")
```




```
set.seed(8)

# Vector of values from Exponential distribution
exp_vec <- rexp(n = 1000, rate = 6)

# Plot of generated values (vector must be dataframe for use with ggplot)
ggplot(data.frame(exp_vec), aes(x = exp_vec, y = ..density..)) +
  geom_histogram(color = "black", fill = "grey") +
  geom_density(size = 1) +
  labs(x = "x",
       title = "Values generated from Exponential distribution",
       subtitle = "Rate = 6")
```

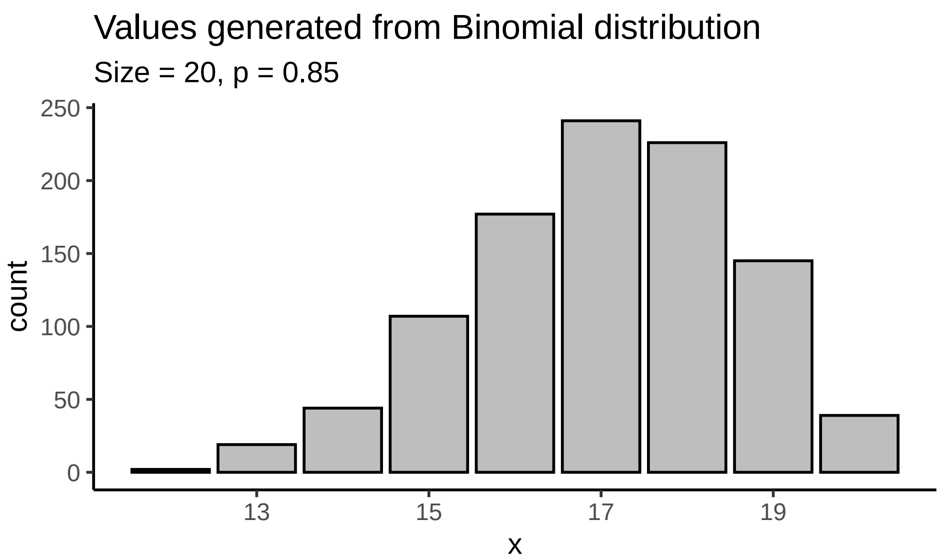



```
set.seed(8)

# Vector of values from Poisson distribution
pois_vec <- rpois(n = 1000, lambda = 10)

# Plot of generated values (vector must be dataframe for use with ggplot)
ggplot(data.frame(pois_vec), aes(x = pois_vec)) +
  geom_bar(color = "black", fill = "grey") +
  labs(x = "x",
       title = "Values generated from Poisson distribution",
       subtitle = "Lambda = 10")
```




```
set.seed(8)

# Vector of values from Binomial distribution
binom_vec <- rbinom(n = 1000, size = 20, p = 0.85)
```

```
# Plot of generated values (vector must be dataframe for use with ggplot)
ggplot(data.frame(binom_vec), aes(x = binom_vec)) +
  geom_bar(color = "black", fill = "grey") +
  labs(x = "x",
       title = "Values generated from Binomial distribution",
       subtitle = "Size = 20, p = 0.85")
```



Bivariate Normal distribution: Joint density plot.

The first code chunk below generates 5000 observations from the bivariate Normal distribution we saw previously. We use a second code chunk to then plot the density of the multivariate normal distribution. Since we are dealing with 2 jointly distributed variables, the joint density is now captured by a surface instead of a curve and must be plotted in 3 dimensions instead of 2. We will use `plot_ly()` to create the 3D plot, but we must use the `kde2d()` function from the **MASS** package first to estimate the density from the generated data.

```
set.seed(86)

# Specify 2-dimensional mean vector
(mean_vector <- c(0, 2))
```

```
## [1] 0 2
```

```
# Specify 2x2 identity matrix as the covariance matrix
(covariance_matrix <- diag(2))
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

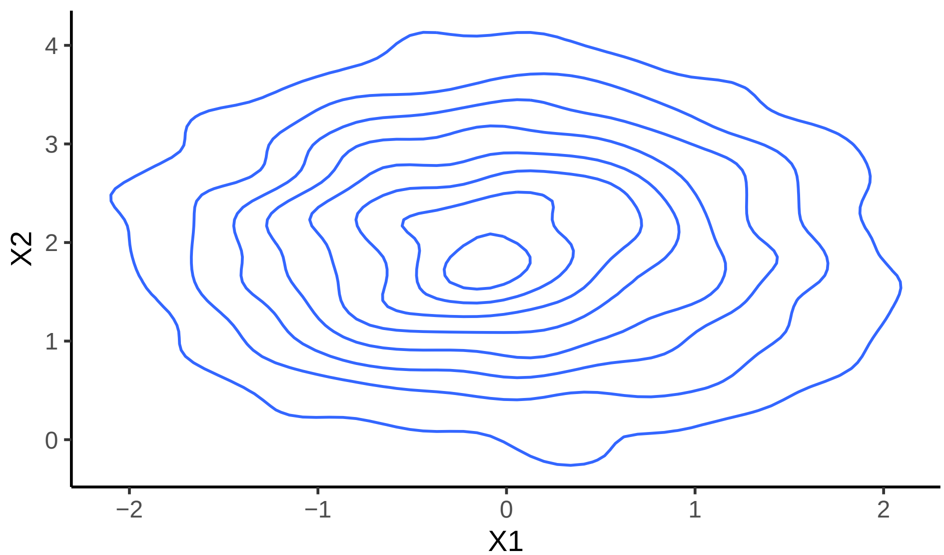
```
# Generate 5000 observations and create data frame for use with ggplot()
mvn_matrix <- MASS::mvrnorm(n = 5000, mu = mean_vector, Sigma = covariance_matrix) %>%
  data.frame()
```

Note: If you run the next chunk and it tells you there is an error displaying the plot, go to the menu options above Tools » Global Options, and then without changing the left side tab, hit "Advanced". Change the Rendering Engine to Desktop OpenGL. This will require a restart of R.

```r
# Estimate kernel density from generated data
mvn_3d <- MASS::kde2d(x = mvn_matrix$X1, y = mvn_matrix$X2)

# Create 3D density plot
plot_ly(x = mvn_3d$x, y = mvn_3d$y, z = mvn_3d$z) %>%
  add_surface()
```

Note: The joint density above can also be represented in 2D using a contour plot, as shown in the code chunk below.

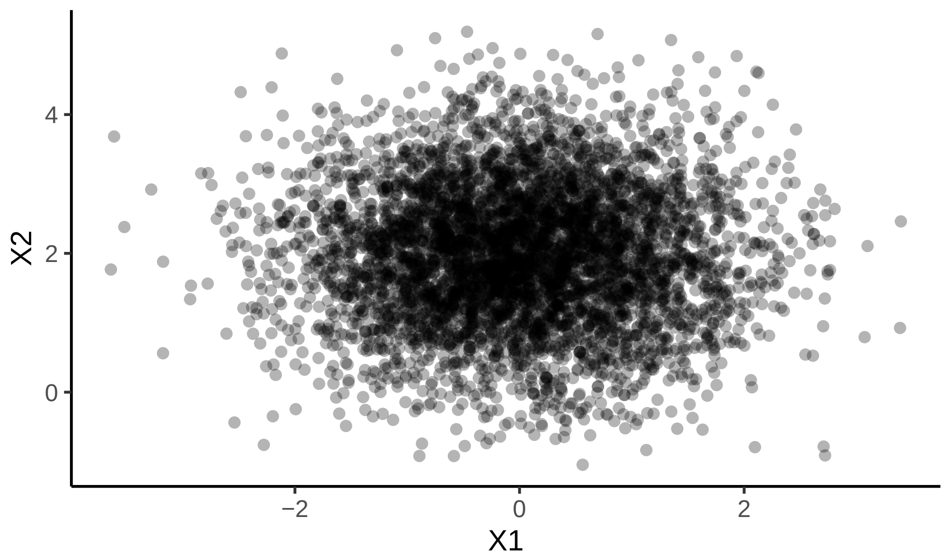
```r
# 2-D contour plot with ggplot
ggplot(mvn_matrix, aes(x = X1, y = X2)) +
  geom_density_2d()
```



part b - Bivariate Normal distribution: Scatterplot. We can further explore the bivariate Normal data we generated by visualizing the data with a scatterplot. Do you think the 2 variables are independent (uncorrelated)?

Solution: Yes, the two variables are uncorrelated. The observed correlation is close enough to 0 that I believe the theoretical correlation in the data generation was 0 (and it was).

```r
# Visualize association
ggplot(mvn_matrix, aes(x = X1, y = X2)) +
  geom_point(alpha = 0.3)
```



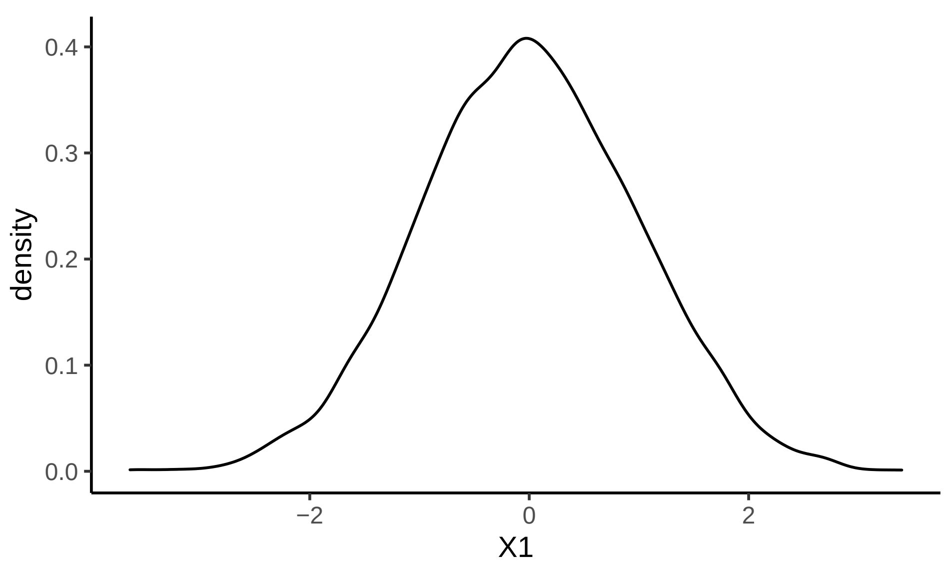

```
# Estimate correlation
summarize(mvn_matrix, cor = cor(X1, X2))
```

```
           cor
1 -0.008646014
```

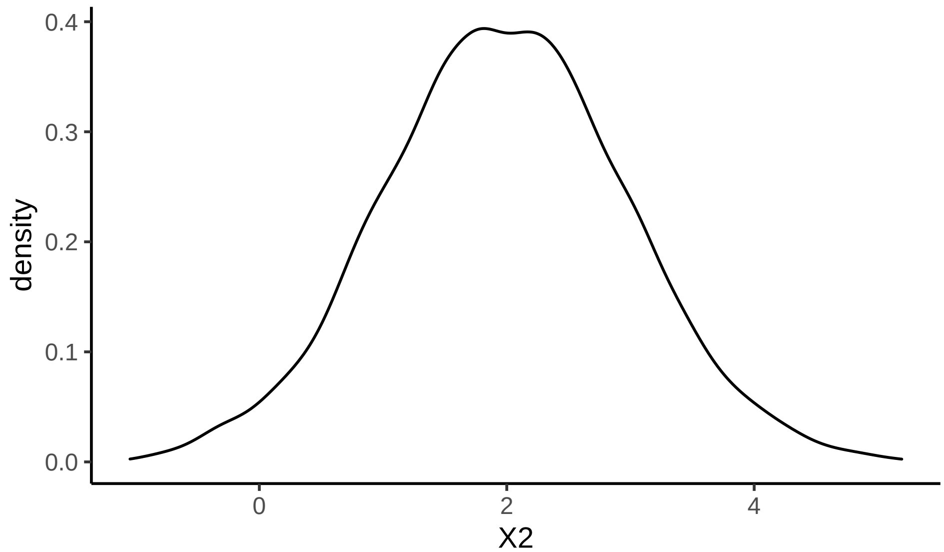
part c - Bivariate Normal distribution: Marginal distributions. In addition to exploring the joint distribution, we can explore the marginal distribution of each variable. Based on the parameters (mean and covariance matrix) we used to generate the multivariate Normal distribution, each marginal distribution should be a Normal distribution. Does it look like this holds?

Solution: Yes, both appear to have normal distributions. The density plots show bell-shaped, unimodal, symmetric curves.

```
# Plot marginal distribution of X1
ggplot(mvn_matrix, aes(x = X1)) +
  geom_density()
```



```
# Plot marginal distribution of X2
ggplot(mvn_matrix, aes(x = X2)) +
  geom_density()
```

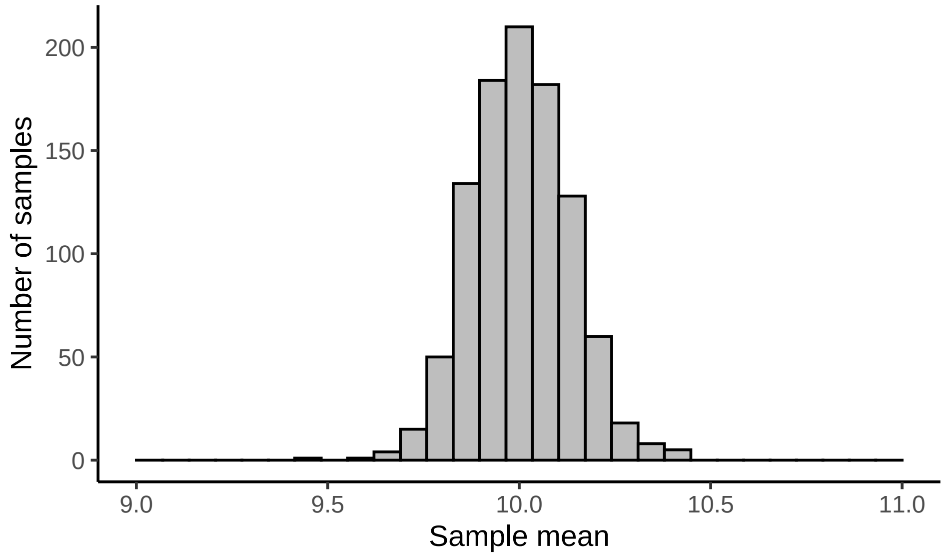


part d - Example: Simulating a sampling distribution of the sample mean. Run the code below to create an empirical sampling distribution of the sample mean for 1000 samples of size 250. What happens to the sampling distribution when the sample size (`n_obs`) goes down to 100? 50? 25?

Solution: We change the n_obs ourselves and observe that the mean of the sampling distribution remains the same as n_obs decreases. However, there is more variability in the distribution as n_obs decreases. (This is the theoretical result described by the Central Limit Theorem (CLT) from intro stats.)

```
set.seed(2021)

# Simulation size
n_sim <- 1000

# Number of observations in each random sample
n_obs <- 250

# Initialize empty vector to store results
means <- rep(NA, n_sim)

for (i in 1:n_sim){
  # Generate sample of Normal data
  dat <- rnorm(n_obs, mean = 10, sd = 2)

  # Compute sample mean
  means[i] <- mean(dat)
}

# Plot empirical sampling distribution of the sample mean
ggplot(data.frame(means), aes(x = means)) +
  geom_histogram(color = "black", fill = "grey") +
  lims(x = c(9, 11)) +
  labs(x = "Sample mean",
```

```
       y = "Number of samples")
```



2 - Setting up simulations

The following problem is modified from* *MDSR Exercise 8.10*.

A research group measured 100 different predictor variables (X_1, ..., X_{100}) on $n = 250$ people, and wants to narrow down the set of variables to include in their final multiple regression model of the outcome (Y). First, they fit 100 different simple linear regression models (i.e., $\hat{Y} = b_0 + b_1X_1$, $\hat{Y} = b_0 + b_1X_2$, ...), and tested the significance of the predictor each time. Then, every predictor that was significant in its simple linear regression model was included in the final multiple regression model. For example, if only X_{23}, X_{47}, and X_{54} were significant out of the 100 simple regression models, then the final model used those variables as predictors, i.e. $Y = b_0 + b_1X_{23} + b_2X_{47} + b_3X_{54}$. This is an example of *p-hacking*, which is both unethical and statistically invalid! We will use this problem to to work on our simulation skills and get a better understanding of why this approach is not statistically valid. Ultimately, we want to use simulation to see what the distribution of the p-value for the overall test of the final multiple regression model looks like, assuming that there are *no* associations between any of the predictors and the outcome (all are assumed to be normally distributed and independent of one another, giving a multivariate normal distribution). We will build up to this final goal over the next two parts of the lab.

Start small.

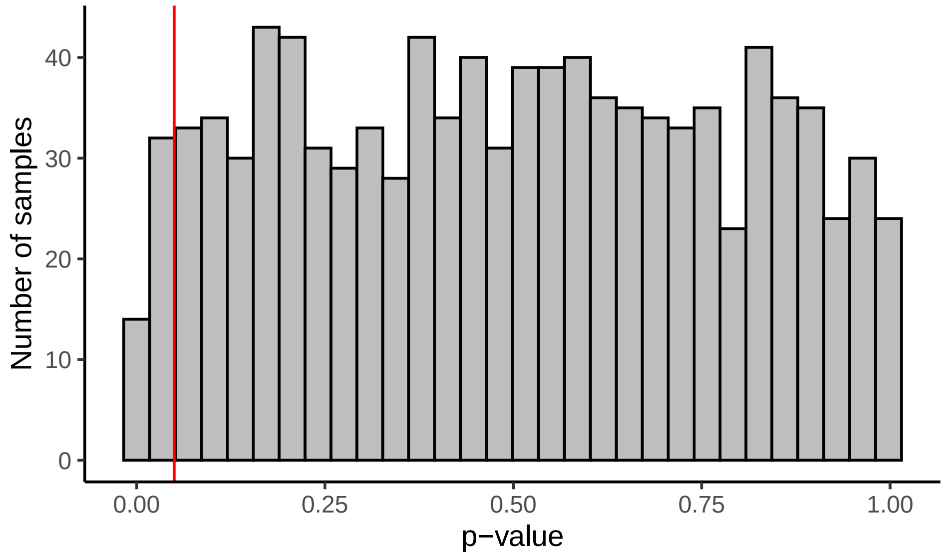
First, let's carry out a simulation that just looks at the distribution of the p-value for the test of a single predictor in a simple linear regression model, assuming there is *no* association between the predictor and the outcome (i.e., generate data assuming the null hypothesis is true). It's helpful to start with a simulation like this, where we *know* what to expect for the result so we can use it as a check to make sure things are set-up/working correctly to start. Because we know the null hypothesis is true, the distribution of the p-values should be uniformly distributed between 0 and 1 (i.e., equally likely to take on any value between 0 and 1).

> part a - If we use a significance level of $\alpha = 0.05$ to conduct each test, what proportion of p-values do we expect will be statistically significant (i.e., what proportion of times will we falsely reject the null hypothesis)? Are our simulated results in line with what we expected?

Solution: We generated data where the null hypothesis is true, so the only time we reject the null should be errors. Since we are using an alpha of 0.05, that's the threshold and should match our error rate. So, we should reject the null about 5 percent of the time, and those would be `false` rejections. The simulation yields a rate of 4.4 percent which is in line with our expectations.

```
# Set the seed for reproducibility!
set.seed(231)

#############################
# Set simulation parameters #
#############################

# Simulation size
n_sim <- 1000

# Number of observations in each random sample
n_obs <- 250

##############################
# Steps for single iteration #
##############################

# Generate predictor
x <- rnorm(n = n_obs, mean = 0, sd = 1)

# Generate outcome (independent of x)
```

```r
y <- rnorm(n = n_obs, mean = 0, sd = 1)

# Fit simple linear regression model
mod <- lm(y ~ x)

# Extract p-value for predictor
pvalue <- mod %>%
  # Use tidy() from broom package to make p-values easy to extract
  broom::tidy() %>%
  filter(term == "x") %>%
  pull(p.value)


##############################################
# Simulate!                                  #
# (repeat many times and summarize results) #
##############################################

# Initialize vector for storing the simulated p-values
pvalues <- rep(NA, n_sim)

for(i in 1:n_sim){
  # Generate predictor
  x <- rnorm(n = n_obs, mean = 0, sd = 1)

  # Generate outcome (independent of x)
  y <- rnorm(n = n_obs, mean = 0, sd = 1)

  # Fit single bivariate model
  mod <- lm(y ~ x)

  # Extract p-value for predictor
  pvalues[i] <- mod %>%
    # Use tidy() from broom package to make p-values easy to extract
    broom::tidy() %>%
    filter(term == "x") %>%
    pull(p.value)
}

# Generate target visualization: histogram of p-values
# When null is true, sampling dist of p-value is Uniform(0, 1)
ggplot(data.frame(pvalues), aes(x = pvalues)) +
  geom_histogram(color = "black", fill = "grey") +
  # Represent significance level cut-off
  geom_vline(xintercept = 0.05, color = "red") +
  labs(x = "p-value", y = "Number of samples")
```




```
# Generate target summary: empirical type I error rate
# Given null is true, should be around 5%
data.frame(pvalues) %>%
  mutate(sig_05 = ifelse(pvalues < 0.05, 1, 0)) %>%
  summarize(n_sim = n(), empirical_t1error = mean(sig_05))
```

```
##   n_sim empirical_t1error
## 1  1000             0.044
```

Then build up…

Now, let's add in 99 more potential predictors, so there are 100 predictors in all. Assume none of the predictors are associated with the outcome OR each other. Assume all predictors and the outcome are from standard Normal distributions (this means mean 0 and standard deviation 1), and use $\alpha = 0.05$ for each test. In this case, we want to know…

1. What is the probability of falsely rejecting H_0 for the first predictor (empirical Type I error rate for X_1)?
2. What is the probability of falsely rejecting H_0 for each of the other 99 predictors on an individual basis (empirical Type I error rate for $X_2, \ldots, X_{100}$)?
3. What is the probability that *at least one of the 100 predictors* is statistically significant at the $\alpha = 0.05$ level?

Note: While we used **tidyverse** in the previous problem to extract p-values, base R is often much faster to run for complex simulations, thus we switch to base R below for extracting p-values so we don't have to wait as long for results.

Another note: You will see a new option below, cache = TRUE for R chunks. This setting tells R to save the results of the chunk, so the chunk doesn't have to re-execute when compiling, as long as you don't change anything within the chunk. This is very useful if you have a simulation that takes a few minutes to run (assuming it is working correctly). You run it once, R saves that, and loads it as needed.

Remember R won't re-execute the chunk as long as you don't change anything in the chunk. This is motivation for running the simulation itself in ONE chunk with cache = TRUE, and then doing any plotting, analysis, etc. of results in ANOTHER chunk (which won't need cache = TRUE). If you keep both in the same chunk, and keep editing your plot title for example, the code will continue to take a long time to run because it keeps re-executing the simulation.

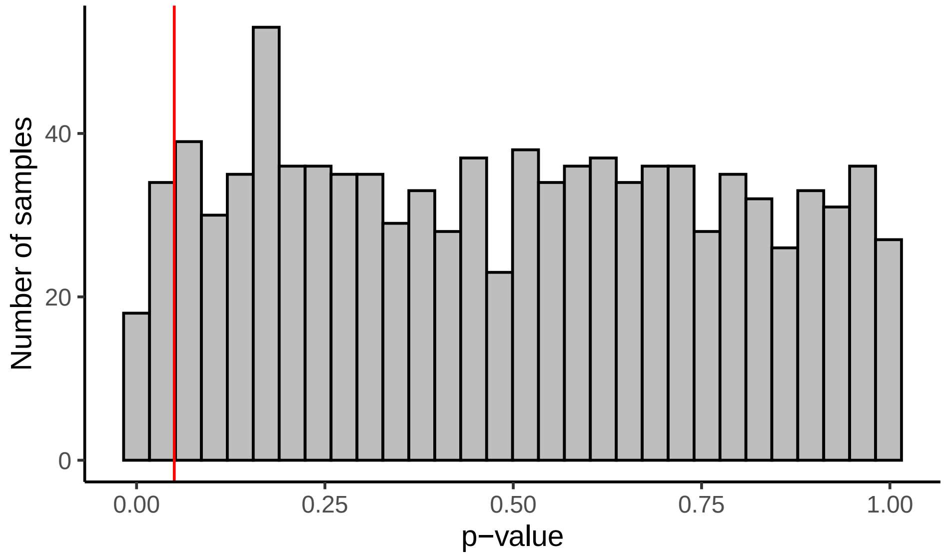
part b - Run the code below to get answers to questions 1-3. Report your findings.

I suggest reviewing the code and trying to determine what every step is doing before running it. It will take a few minutes to run.

Solution:

```
# Set the seed for reproducibility
set.seed(231)

##############################
# Set simulation parameters #
##############################

# Simulation size
n_sim <- 1000

# Number of observations in each random sample
n_obs <- 250

# Number of predictor variables
n_x <- 100

###############################
# Steps for single iteration #
###############################

# Generate 100 predictors from multivariate Normal distribution
xs <- MASS::mvrnorm(n = n_obs, mu = rep(0, n_x), Sigma = diag(n_x)) %>%
  data.frame()

# Generate outcome (independent of the 100 predictors)
dat <- xs %>%
  mutate(y = rnorm(n = n_obs, mean = 0, sd = 1))

# Fit 100 different simple linear regression models (one model for each predictor)
# and extract p-values
pvalues <- rep(NA, n_x)
for (j in 1:n_x){
  # Fit model
  mod <- lm(formula = paste0("y ~ X", j), data = dat)

  # Extract p-value for predictor (tidy)
  # pvalues[j] <- mod %>%
  #   # Use tidy() from broom package to make p-values easy to extract
  #   broom::tidy() %>%
  #   filter(term == paste0("X", j)) %>%
  #   pull(p.value)

  # Extract p-value (base R is faster in this case)
  pvalues[j] <- (summary(mod)$coeff)[paste0("X", j), "Pr(>|t|)"]
}

#############################################
# Simulate!                                 #
```

```r
# (repeat many times and summarize results) #
#############################################

# Initialize matrix for storing the p-values
# - Columns will be predictors X1-X100
# - Rows will be the different iterations, 1 per n_sim
# - There are two `for` loops, notice how we used j for the index above?
pvalues <- array(NA, dim = c(n_sim, n_x)) %>% data.frame()

for (i in 1:n_sim){
  # Generate 100 predictors from multivariate Normal distribution
  xs <- MASS::mvrnorm(n = n_obs, mu = rep(0, n_x), Sigma = diag(n_x)) %>%
    data.frame()

  # Generate outcome (independent of the 100 predictors)
  dat <- xs %>%
    mutate(y = rnorm(n = n_obs, mean = 0, sd = 1))

  # Fit 100 different bivariate models (one model for each predictor)
  # and extract p-values
  for (j in 1:n_x){
    # Fit model
    mod <- lm(formula = paste0("y ~ X", j), data = dat)

    # Extract p-value using base R method
    pvalues[i, j] <- (summary(mod)$coeff)[paste0("X", j), "Pr(>|t|)"]
  }
}

# Generate target visualization for Q1: histogram of p-values for X1
# - When null is true, sampling dist of p-value is Uniform(0, 1)
ggplot(data.frame(pvalues), aes(x = X1)) +
  geom_histogram(color = "black", fill = "grey") +
  # Represent significance level cut-off
  geom_vline(xintercept = 0.05, color = "red") +
  labs(x = "p-value", y = "Number of samples")
```




```
# Generate target summary for Q2: empirical type I error rates for each predictor
# - Given null is true, should be around 5% each
pvalues <- data.frame(pvalues) %>%
  # check whether each p-value is less than 0.05 (significant = 1), and
  # use across() to repeat check across every column
  mutate(across(everything(), ~ ifelse(. < 0.05, 1, 0), .names = "{.col}_sig"))

(error_rates <- pvalues %>%
  # compute proportion of p-values that are significant for every column
  summarize(across(ends_with("_sig"), mean)))
```

```
  X1_sig X2_sig X3_sig X4_sig X5_sig X6_sig X7_sig X8_sig X9_sig X10_sig
1  0.048  0.053  0.049  0.039  0.033  0.068  0.054  0.045  0.033   0.047
  X11_sig X12_sig X13_sig X14_sig X15_sig X16_sig X17_sig X18_sig X19_sig
1   0.042   0.054   0.037   0.058   0.057   0.056   0.055   0.049   0.046
  X20_sig X21_sig X22_sig X23_sig X24_sig X25_sig X26_sig X27_sig X28_sig
1   0.055   0.046   0.064   0.049   0.045   0.059   0.054    0.05   0.061
  X29_sig X30_sig X31_sig X32_sig X33_sig X34_sig X35_sig X36_sig X37_sig
1   0.046   0.059   0.053   0.045   0.053   0.043   0.037   0.046   0.035
  X38_sig X39_sig X40_sig X41_sig X42_sig X43_sig X44_sig X45_sig X46_sig
1   0.047   0.048   0.059   0.054   0.059   0.038   0.044   0.049   0.054
  X47_sig X48_sig X49_sig X50_sig X51_sig X52_sig X53_sig X54_sig X55_sig
1   0.027   0.035   0.047   0.055   0.053    0.05   0.059   0.041   0.053
  X56_sig X57_sig X58_sig X59_sig X60_sig X61_sig X62_sig X63_sig X64_sig
1   0.046   0.048   0.042   0.044   0.046   0.051   0.062   0.067   0.046
  X65_sig X66_sig X67_sig X68_sig X69_sig X70_sig X71_sig X72_sig X73_sig
1   0.055   0.054   0.048    0.05   0.061   0.053   0.055   0.043    0.04
  X74_sig X75_sig X76_sig X77_sig X78_sig X79_sig X80_sig X81_sig X82_sig
1   0.064   0.047    0.06   0.042   0.057   0.043    0.05   0.053   0.049
  X83_sig X84_sig X85_sig X86_sig X87_sig X88_sig X89_sig X90_sig X91_sig
1   0.052   0.051   0.051   0.054   0.052    0.05   0.054   0.062   0.044
  X92_sig X93_sig X94_sig X95_sig X96_sig X97_sig X98_sig X99_sig X100_sig
1   0.041   0.051   0.055    0.05   0.047   0.048   0.051   0.046    0.049
```

```
# Generate target summary for Q3: probability at least 1 of 100 predictors is significant
# - each row represents one iteration;
```

```r
# - need to see how many rows have at least one significant p-value
# - expect P(at least one) = 1 - P(none) = 1 - (0.95)^100 = 0.994
pvalues %>%
  # must invoke `rowwise()` in order to use `sum()` across rows
  rowwise() %>%
  # check if the total number significant in each row is greater than 0
  mutate(sum_sig = sum(c_across(ends_with("_sig"))),
         any_sig = ifelse(sum_sig > 0, 1, 0)) %>%
  # stop row-wise calculations
  ungroup() %>%
  # compute proportion of rows with at least one significant p-value
  summarize(n_sim = n(),
            prop_sims_sig = mean(any_sig))
```

```
# A tibble: 1 x 2
  n_sim prop_sims_sig
  <int>         <dbl>
1  1000         0.987
```

We are tasked with answering the following based on the simulation above:

1. What is the probability of falsely rejecting H_0 for the first predictor (empirical Type I error rate for X_1)?

In our simulation, we obtained X1_sig of 0.048, based on the output.

2. What is the probability of falsely rejecting H_0 for each of the other 99 predictors on an individual basis (empirical Type I error rate for $X_2, \ldots, X_{100}$)?

These values are contained in X2_sig to X100_sig, and are generally around 0.05 each.

3. What is the probability that *at least one of the 100 predictors* is statistically significant at the $\alpha = 0.05$ level?

In our 1000 simulations, 987 resulted in at least one predictor being significant out of the 100. So, the probability is 0.987. However, we know there is no relationship present, so any indication of significance is incorrect.

This is why we don't advocate running a regression with 100 or even 300 predictors. You are likely to find "significance" but is it real?

3 - . . . and put it all together!

Now, follow the process the researchers took. As a reminder, here's the problem statement:

A research group measured 100 different predictor variables (X_1, . . . , X_{100}) on $n = 250$ people, and wants to narrow down the set of variables to include in their final multiple regression model of the outcome (Y). First, they fit 100 different simple linear regression models (i.e., $\hat{Y} = b_0 + b_1 X_1$, $\hat{Y} = b_0 + b_1 X_2$, . . .), and tested the significance of the predictor each time. Then, every predictor that was significant in its simple linear regression model was included in the final multiple regression model. For example, if only X_{23}, X_{47}, and X_{54} were significant out of the 100 simple regression models, then the final model used those variables as predictors, i.e. $Y = b_0 + b_1 X_{23} + b_2 X_{47} + b_3 X_{54}$.

What does the distribution of the p-value for the overall test of the final multiple regression model look like, assuming that there are no associations between any of the predictors and the outcome (all are assumed to be multivariate normal and independent of one another). What does this tell us about the problem with p-hacking?

Note: The *overall test* means the overall F test of H_0 : *all* $\beta_j = 0$, vs. H_a : *at least one* $\beta_j \neq 0$. Note that you can extract this p-value for the overall F-test using the code `glance(model)$p.value`, where the `glance()` function is from the **broom** package. For example, if we fit 100 different simple linear regression models and only the models with X_{23}, X_{47}, and X_{54} turned out to have significant p-values, then we would do the following:

```
# fit example multiple regression
test_model <- lm(y ~ X23 + X47 + X54, data = dat)
mosaic::msummary(test_model)
```

```
               Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.04456    0.06485   0.687    0.493
X23          0.07301    0.06170   1.183    0.238
X47         -0.04135    0.06115  -0.676    0.500
X54          0.10601    0.06491   1.633    0.104

Residual standard error: 1.017 on 246 degrees of freedom
Multiple R-squared:  0.01747,  Adjusted R-squared:  0.005492
F-statistic: 1.458 on 3 and 246 DF,  p-value: 0.2265
```

```
# Confirm p-value matches the p-value on the last line of the summary output
broom::glance(test_model)$p.value
```

```
    value
0.2265094
```

If you are not familiar with this test procedure, that's okay. Think of it as testing `in bulk` - it looks to see whether any predictor(s) in the model are 'useful' for predicting the response (meaning will have a slope significantly different from 0). The null hypothesis is that none of the predictors are useful, while the alternative is that at least one is. In trade for testing in `bulk`, it won't tell you which predictors were 'useful'. The F-test is often thus used in conjunction with the individual t-tests. You can learn more about this procedure in Stat 230.

> part a - Before beginning to code, think about the last simulation we carried out. What steps need to be added/removed/modified in that simulation to carry out this simulation?

Solution:

The data-generating process is the same as before, so nothing needs to be updated in the steps used to generate the random predictors and outcome. The first step in the procedure described above ("Fit each of the 100 bivariate models") is also what we did in the above simulation, so we can keep that code, but we need to modify it to pull out which variables are statistically significant ($p < 0.05$). Then, we need to add code to fit a model that includes each of the significant predictors identified via the bivariate models. Lastly, we'll want to extract the p-value from the overall F-test for that overall model.

> part b - Write code to walk through each of the general steps for one iteration (i.e., outside of a `for()` loop). *Hint:* For a single vector of p-values, the function `which()` can be used to identify the indices of values that are less than 0.05 (e.g., `which(pvalues < 0.05)` might return "23, 47, 54, …"). We can then use `paste0()` to piece together predictor names (e.g., to get `"X23"`, `"X47"`, …), and another layer of `paste(..., collapse = " + ")` to combine all the predictor names into the right-hand side of the linear model formula (e.g., to get `"X23 + X47 + ..."`).

Solution:

```
##############################
# Steps for single iteration #
##############################

# Generate 100 predictors from multivariate Normal distribution
xs <- MASS::mvrnorm(n = n_obs, mu = rep(0, n_x), Sigma = diag(n_x)) %>%
  data.frame()

# Generate outcome (independent of the 100 predictors)
dat <- xs %>%
  mutate(y = rnorm(n = n_obs, mean = 0, sd = 1))

# Fit 100 different bivariate models (one model for each predictor)
# and extract p-values
pvalues <- rep(NA, n_x)

for (j in 1:n_x){
  # Fit model
  mod <- lm(formula = paste0("y ~ X", j), data = dat)
  # Extract p-value using base R method
  pvalues[j] <- (summary(mod)$coeff)[paste0("X", j), "Pr(>|t|)"]
}

# Grab predictors with significant p-values and paste together
sig_predictors <- paste(paste0("X", which(pvalues < 0.05)), collapse = " + ")

# Fit overall model
mod_overall <- lm(formula = paste("y ~ ", sig_predictors), data = dat)

# Extract overall p-value
broom::glance(mod_overall)$p.value
```

```
        value
0.00003198467
```

> part c - Now incorporate your code above into a `for()` loop to iterate through 1000 simulations. Be sure to save the p-values from the overall F-tests.

Solution:

```
#Set the seed for reproducibility!
set.seed(231)

##############################
# Set simulation parameters #
```

```r
##############################
# Simulation size
n_sim <- 1000
# Number of observations in each random sample
n_obs <- 250
# Number of predictor variables
n_x <- 100


##############################################
# Simulate! #
# (repeat many times and summarize results) #
##############################################
# Initialize vector for storing the overall p-values
pvalues_overall <- rep(NA, n_sim)

for(i in 1:n_sim){
# Generate 100 predictors from multivariate Normal distribution
xs <- MASS::mvrnorm(n = n_obs, mu = rep(0, n_x), Sigma = diag(n_x)) %>%
  data.frame()

# Generate outcome (independent of the 100 predictors)
dat <- xs %>%
  mutate(y = rnorm(n = n_obs, mean = 0, sd = 1))

# Fit 100 different bivariate models (one model for each predictor)
# and extract p-values
pvalues <- rep(NA, n_x)

  for (j in 1:n_x){
    # Fit model
    mod <- lm(formula = paste0("y ~ X", j), data = dat)
    # Extract p-value (base R is faster in this case)
    pvalues[j] <- (summary(mod)$coeff)[paste0("X", j), "Pr(>|t|)"]
  }

  # Find indices of pvalues that are less than 0.05
  sig_indices <- which(pvalues < 0.05)

  # If there are no significant predictors, set overall p-value to NA
  # Otherwise, fit overall model and grab overall p-value
  if(length(sig_indices) == 0){
  # cannot fit final regression model
  pvalues_overall[i] <- NA
  } else{
  # Grab predictors with significant p-values and paste together
  sig_predictors <- paste(paste0("X", sig_indices), collapse = " + ")
  # Fit overall model
  mod_overall <- lm(formula = paste("y ~ ", sig_predictors), data = dat)
  # Extract overall p-value
  pvalues_overall[i] <- broom::glance(mod_overall)$p.value
  }
}
```

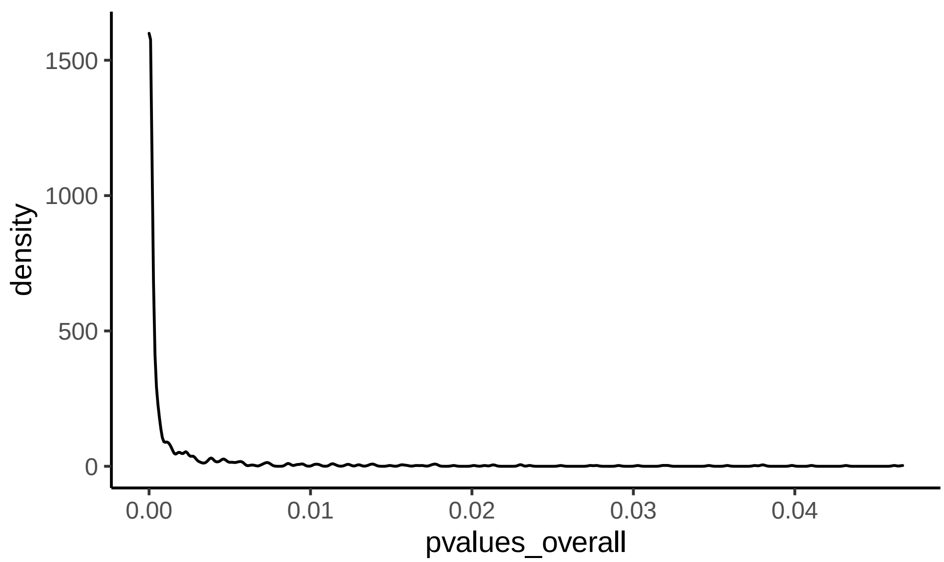
part d - Create a histogram or density plot of the sampling distribution of the p-values from the overall test. What do you notice about the distribution? What proportion of the overall tests are significant at the 0.05 level? What about the 0.01 level? What about the 0.001 level? What does this tell you about the credibility of this approach to model-building, and thus one of the (many) problems with p-hacking? Do you think this is ethical?

Solution:

The density plot shows a very right-skewed distribution with the vast majority of p-values being less than 0.01. We see below that 98.7% of the p-values were less than 0.05, and 75.8% were less than 0.001. Over 50% of the p-values were less than 0.0001! That sure would perpetuate some false credibility/confidence in this result if the authors only present the overall model and don't describe their process.

Even when there are NO true associations in the underlying relationship between any of the predictors and the outcome, using this process as described will almost surely result in a highly statistically significant overall model. This is statistically invalid since we know none of the predictors are associated with the outcome in the first place, and thus incredibly unethical if results from the final model are reported as significant.

```
ggplot(data = data.frame(pvalues_overall), aes(x = pvalues_overall)) +
  geom_density()
```




```
data.frame(pvalues_overall) %>%
  filter(!is.na(pvalues_overall)) %>%
  mutate(sig05 = ifelse(pvalues_overall < 0.05, 1, 0),
         sig001 = ifelse(pvalues_overall < 0.001, 1, 0),
         sig0001 = ifelse(pvalues_overall < 0.0001, 1, 0)) %>%
  summarize(# proportion of overall tests significant at 0.05 level
            prop05 = sum(sig05)/n_sim,
            # proportion of overall tests significant at 0.001 level
            prop001 = sum(sig001)/n_sim,
            # proportion of overall tests significant at 0.0001 level
            prop0001 = sum(sig0001)/n_sim)
```

```
  prop05 prop001 prop0001
1  0.987   0.758    0.522
```

4 - Computational Time

As you develop more complex simulations, efficiency in code can become more important. The difference of a few seconds between two methods to do the same thing can result in a simulation that takes one hour or two hours, depending on the simulation size.

The function `microbenchmark()` from the **microbenchmark** package measures the time it takes to evaluate certain code, and can be useful to compare the execution time of different expressions. By default, `microbenchmark()` runs each argument 100 times. It then returns summary statistics (min, mean, median, max) on the run time for each expression.

Example: in the lab, we used the `mvrnorm()` function from the **MASS** package to generate values from a multivariate normal distribution. The same type of function, `rmvnorm()`, is available in the **mvtnorm** package. Does one of these functions execute faster than the other?

```
set.seed(2021)
microbenchmark(MASS::mvrnorm(n = n_obs, mu = rep(0, n_x), Sigma = diag(n_x)),
               mvtnorm::rmvnorm(n = n_obs, mean = rep(0, n_x), sigma = diag(n_x)))
```

```
Unit: milliseconds
                                                                  expr    min
      MASS::mvrnorm(n = n_obs, mu = rep(0, n_x), Sigma = diag(n_x)) 3.8895
 mvtnorm::rmvnorm(n = n_obs, mean = rep(0, n_x), sigma = diag(n_x)) 4.3655
      lq     mean  median      uq     max neval
 3.99055 4.276859 4.09335 4.34675  7.0503   100
 4.46370 4.828222 4.57970 4.84790 12.6458   100
```

```
# update to execute 200 times
microbenchmark(MASS::mvrnorm(n = n_obs, mu = rep(0, n_x), Sigma = diag(n_x)),
               mvtnorm::rmvnorm(n = n_obs, mean = rep(0, n_x), sigma = diag(n_x)),
               times = 200)
```

```
Unit: milliseconds
                                                                  expr    min
      MASS::mvrnorm(n = n_obs, mu = rep(0, n_x), Sigma = diag(n_x)) 3.8907
 mvtnorm::rmvnorm(n = n_obs, mean = rep(0, n_x), sigma = diag(n_x)) 4.3462
     lq     mean  median      uq    max neval
 3.9659 4.184094 4.05675 4.22155 7.2084   200
 4.4451 4.753959 4.56440 4.78985 7.9355   200
```

```
# add names so the output is easier to read
microbenchmark(mvrnorm = MASS::mvrnorm(n = n_obs, mu = rep(0, n_x), Sigma = diag(n_x)),
               rmvnorm = mvtnorm::rmvnorm(n = n_obs, mean = rep(0, n_x), sigma = diag(n_x)),
               times = 200)
```

```
Unit: milliseconds
    expr    min      lq     mean  median      uq    max neval
 mvrnorm 3.8872 3.97205 4.193756 4.07565 4.23505 6.9543   200
 rmvnorm 4.3422 4.47110 4.787161 4.57505 4.80700 7.9650   200
```

> See if you can write a function and use either one of the `apply()` functions or one of the newer `map()`ping functions (e.g. `pmap_dfr()`; see MDSR Chapter 7 and MDSR Chapter 13) to run your simulation instead of using a `for()` loop. Does it execute faster?

Solution:

Answers will vary.