Analyzing Salaries in the AI Job Market

Andrew Mitchell

Sampling Exercise Review

In class we wrote some code to analyse our samples in a `.R` file named `sampling-exercise.R`. This is useful to save and quickly run a full script (set of many lines of code to perform one or many actions). You can take a look at a completed version of the file we looked at in class which I have written.

`.R` files are the basis of saving, running, and sharing R code - this is what all of the packages we've been loading and using functions from use to write their code. They let you move out of the console where each line is temporary and begin to build up more complicated workflows and analyses.

In this file, we will take the next step into a system called Quarto. Whereas `.R` files are pure R code - everything written in them will execute as a command when you run it - a Quarto `.qmd` file allows you to mix together R code, text, and outputs like tables and plots. This is how we can actually communicate our analysis!

Outline

In this page, we will:

1. Review what an `.R` file is and how to use it.
2. Introduce Quarto documents `.qmd` and use one to step through and explain the sampling exercise code we wrote in class.
3. Show off what Quarto can do by extending our sampling code.

Saving and running code in a `.R` script

Before diving into how a Quarto document works, let's review how `.R` files work:

A `.R` script works very much like running single lines of code in the console - it will run each line in order from top to bottom. If you run it from RStudio it will even echo the lines of code into the console so you can track exactly what is happening.

By saving your code in a `.R` file you can:

1. Save your code rather than needing to type it or copy/paste it into the console line by line.
2. Build up a full script to perform several actions at once.

- For instance, to prepare the AI Jobs dataset we looked at in class, I wrote a script to download the data, clean it, and make some adjustments, then save it to a .csv file. Rather than write this each time, I can just `source` the `ai-jobs-data.R` script to do it all at once.
- You can see all the steps of that code in the `ai-jobs-data.R` file. Try to look through the code and identify the blocks of code that logically fit together - in other words, the multiple lines of code which are grouped together because they do the same thing or because they form one step of the process. I have added comments to outline these for you. Even if you don't know what every line or function does, you should be able to follow the logical flow of what the code does.

So, in our `sampling-exercise.R` file we completed a few logical steps:

1. First, we need to load the `tidyverse` library in order to access our standard data processing functions:

   ```
   library(tidyverse)
   ```

2. Then, we input the observations from each sample we took and saved them to a `sample_x` variable. These are all lists (using `c()`) of 10 numbers.

   ```
   sample_1 <- c(83.2, 82.6, 82.6, 82.6, 93.2, 94, 94, 48.5, 33.6, 33.6)
   sample_2 <- c(88, 48, 23, 23, 23, 23, 23, 23, 23, 23)
   # etc. ...
   ```

3. Next, we calculated the means for each sample. I showed two ways of doing this, let's look at just the first for now.

   We calculated the mean of the list of numbers of each sample:

   ```
   mean_1 <- mean(sample_1)
   mean_2 <- mean(sample_2)
   # etc. ...
   ```

4. Then we put these means together into a table:

   ```
   sample_means <- tibble(mean_values = c(mean_1, mean_2, ...))
   ```

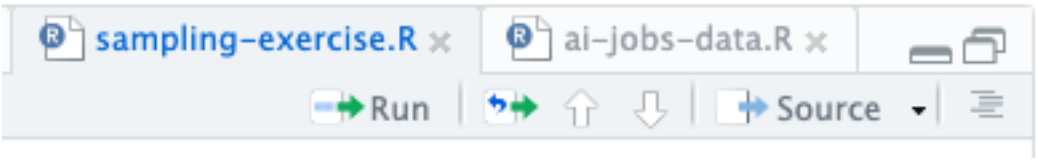
   This gives us a table with one row for each sample we took and a column named `mean_values` which contains the means we calculated:

	mean_values
1	72.8
2	32
...	...

This is where we stopped in class, but compare the steps I just outlined with the code you wrote in your own `.R` file and you should be able to identify these logical blocks of code.

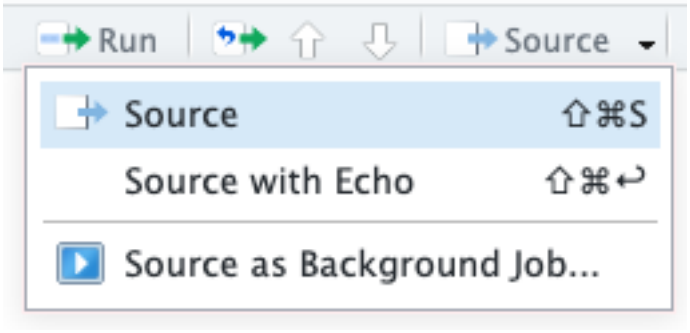
By 'sourcing' the file, we can run this all at once and either print out our `sample_means` table, or take a look at it within RStudio.

Workflow for a `.R` file



RStudio gives you some useful tools for when you are writing an `.R` file.

- To run the whole file (what we call 'source'-ing the file, you can press the 'Source' button at the top right.
  - By default, this will print out just the filepath to your console (e.g. `source("~/Documents/UCL/Teaching/BSSC0021_25/Code/Week4/sampling-exercise.R")`) and display any outputs like plots.
  - If you select the arrow next to it, you can choose 'Source with Echo'. This will print out each row of code to the console (or 'echo' it) as the code runs. This is useful if you want to check exactly what is happening.



- By selecting only certain rows and clicking the 'Run' button, RStudio will run only those lines of code. This is very useful as you are writing your code and building up a full script. You can check what each part does as you go without needing to run the whole file at once.

A suggested workflow for writing a script is to move back and forth between the `.R` file editor and the console. Build your code up in the `.R` file and run each logical chunk of code as you write it to make sure it works the way you expect.

Anything that you need to run which is temporary or a one-off, type this directly in the console (like if you need to look at the help page for a function such as `mean`, you would run `?mean` in the console to bring it up).

Hint: To repeat a previously run line of code in the console, press the up arrow - this will cycle back through the history of commands you have run. Once you get to the one you want, just press enter to run it again.

Once you have a few chunks of code in the `.R` file, you can run 'Source' to check that the whole thing works top to bottom.

Communicating Analysis

In the next section we will introduce Quarto, a system for creating professional looking documents with R. Quarto gives us the power to perform and communicate our analysis, all in one! It's also how I've written all of the presentations and notes for this module (including this page).

We'll introduce it by writing a Quarto document which explains each line of our sampling exercise code.

Appendix

`sampling-exercise.R`:

```
# Load the tidyverse package
# This will provide us with functions like `tibble`, `gather`, `group_by`,
`summarise`, `ggplot`, etc.
library(tidyverse)

# Input the values of the samples

sample_1 <- c(83.2, 82.6, 82.6, 82.6, 93.2, 94, 94, 48.5, 33.6, 33.6)
sample_2 <- c(48.8, 86.5, 67.5, 84.5, 97.6, 92, 60.7, 108, 84.3, 58.5)
sample_3 <- c(128, 53.7, 70.9, 75.2, 84.9, 91.2, 70.2, 82, 88.8, 82)
sample_4 <- c(122, 54, 101, 93.2, 89.4, 64.9, 68.3, 97.7, 77.7, 123)
sample_5 <- c(92.2, 82, 97.7, 48.5, 94, 70.6, 105, 60.7, 65.1, 82.3)
sample_6 <- c(91.2, 110, 57.7, 48.4, 122, 65.5, 86.5, 62.7, 62.7, 85.6)
sample_7 <- c(87.5, 72.8, 84.8, 56.5, 64.9, 42.2, 62.8, 54.1, 84.4, 89.7)
sample_8 <- c(78.8, 48.5, 54, 70.4, 43.8, 65.5, 78.8, 43.8, 113, 123)
sample_9 <- c(89.5, 125, 94, 92.4, 70, 99, 111, 96.9, 64.2, 63.7)
sample_10 <- c(102, 90.8, 110, 123, 79.4, 77.9, 82.3, 92.6, 90.8, 113)
sample_11 <- c(86.6, 83, 58.9, 101, 54.9, 96.8, 84.8, 60.4, 84, 83.3)
sample_12 <- c(63.7, 102, 84.3, 54.1, 71.6, 122, 40.8, 63.7, 84.4, 90.9)
sample_13 <- c(123, 93.5, 68.3, 97.3, 53.1, 50.2, 130, 48.5, 56.5, 65.1)
sample_14 <- c(109, 85.6, 54.9, 62.7, 30.3, 87.1, 94, 111, 54.1, 54.3)
sample_15 <- c(109, 44.2, 99.3, 58.5, 77, 86.2, 125, 128, 79.1, 98.2)
sample_16 <- c(41.8, 122, 70.6, 86.3, 83.7, 84.3, 82.9, 41.3, 72.7, 98.2)
sample_17 <- c(63.7, 89.8, 101, 70.2, 68.4, 77.9, 105, 53.1, 112, 55.8)
sample_18 <- c(48.8, 86.5, 67.5, 84.5, 97.6, 92, 60.7, 108, 84.3, 58.5)
sample_19 <- c(92.6, 82.3, 57.1, 57.1, 82, 62.7, 62.7, 82, 82, 77)
sample_20 <- c(67, 64.5, 87.8, 84.5, 43.8, 67.2, 30.3, 107, 85.2, 67.2)
sample_21 <- c(66.8, 85.6, 123, 109, 97.7, 89.7, 51.9, 48.5, 81.7, 51.9)
sample_22 <- c(97.1, 85.2, 118, 56.5, 91.2, 91.8, 59.3, 111, 93.2, 75.9)
```

```r
sample_23 <- c(111, 64.4, 137, 82.3, 94, 64.9, 39, 75.2, 91.2, 63.8)
sample_24 <- c(97.7, 50.2, 78.8, 119, 109, 119, 40.8, 72.8, 97.1, 102)
sample_25 <- c(58.9, 79.8, 91.8, 56.5, 46.1, 84.4, 71.2, 71.2, 86.2, 70.6)
sample_26 <- c(75.8, 73.8, 57.7, 129, 101, 71.2, 137, 135, 60.3, 77)
sample_27 <- c(56.5, 52.3, 99.3, 107, 75.9, 89.2, 70, 84.3, 66.8, 93.5)
sample_28 <- c(28.3, 90.8, 39, 82.3, 89.5, 58.1, 120, 74.7, 28.8, 117)
sample_29 <- c(97.7, 85.4, 91.1, 54.1, 109, 102, 82, 90.8, 63.7, 92.3)
sample_30 <- c(101, 68.3, 62.8, 65.5, 75.8, 93.8, 81.7, 72.8, 63.4, 109)

# Create a table with the samples and an id column

data <- tibble(
  sample_1, sample_2, sample_3, sample_4, sample_5,
  sample_6, sample_7, sample_8, sample_9, sample_10,
  sample_11, sample_12, sample_13, sample_14, sample_15,
  sample_16, sample_17, sample_18, sample_19, sample_20,
  sample_21, sample_22, sample_23, sample_24, sample_25,
  sample_26, sample_27, sample_28, sample_29, sample_30
) |>
  gather(key = "sample", value = "value")

data # Look at the table

# Calculate the mean of each sample

sample_means <- data |>
  group_by(sample) |>
  summarise(mean = mean(value))

sample_means # Look at the means
#
# # Plot the sampling distribution
#
ggplot(sample_means, mapping = aes(x = mean)) +
  geom_histogram() +
  theme_minimal()
```

Using Quarto for Statistical Analysis

Now that we've seen how `.R` files help us save and organize our code, let's explore how Quarto can make our analysis even better. When we write code in an `.R` file, we often find ourselves adding comments to explain what each part does. These comments help us remember our thinking and help others understand our code. But what if we could write proper explanations, include our code, and show the results all in one place?

This is exactly what Quarto lets us do. Think of it as a document where we can write explanations in normal text, include our R code in special sections called "chunks", and show the output of

that code (like tables and plots) right where we need it. This makes it much easier to explain our analysis to others - or even to ourselves when we come back to it later!

Converting our Sampling Exercise

Let's take the sampling exercise we did in class and see how we can make it clearer using Quarto. We'll take the same code from our `.R` file but now we can properly explain what each part does.

First, just like in our `.R` file, we need to load our tidyverse package. In Quarto, we put R code in special sections marked with three backticks and `{r}`. These are called "code chunks":

```r
library(tidyverse)
```

Now we can input our sample data. Remember these are the measurements we took in class:

```r
sample_1 <- c(83.2, 82.6, 82.6, 82.6, 93.2, 94, 94, 48.5, 33.6, 33.6)
sample_2 <- c(48.8, 86.5, 67.5, 84.5, 97.6, 92, 60.7, 108, 84.3, 58.5)
sample_3 <- c(128, 53.7, 70.9, 75.2, 84.9, 91.2, 70.2, 82, 88.8, 82)
```

I've just shown three samples here to keep things clear, but you can add all of your samples just like we did in the `.R` file. If we ask R to output one of our samples, it will print out within the document:

```r
sample_2
```

```
 [1]  48.8  86.5  67.5  84.5  97.6  92.0  60.7 108.0  84.3  58.5
```

If you are viewing the `.qmd` document itself (i.e. not the rendered version) try running these code blocks by pressing the 'run' button (looks like a triangular play button).

One nice thing about Quarto is that we can explain our thinking right alongside our code. For instance, we can explain that each sample contains 10 measurements, and we're storing them in variables named `sample_1`, `sample_2`, etc.

Now let's calculate the mean for each sample:

```r
mean_1 <- mean(sample_1)
mean_2 <- mean(sample_2)
mean_3 <- mean(sample_3)
```

We can even show the results right in our text! For example, the mean of our first sample is 72.79 (this is printed by just putting `{r} mean_1` inline) This is much nicer than having to print values separately - we can discuss the numbers right where they make sense in our explanation.

Next, let's create our table of means just like we did in the `.R` file:

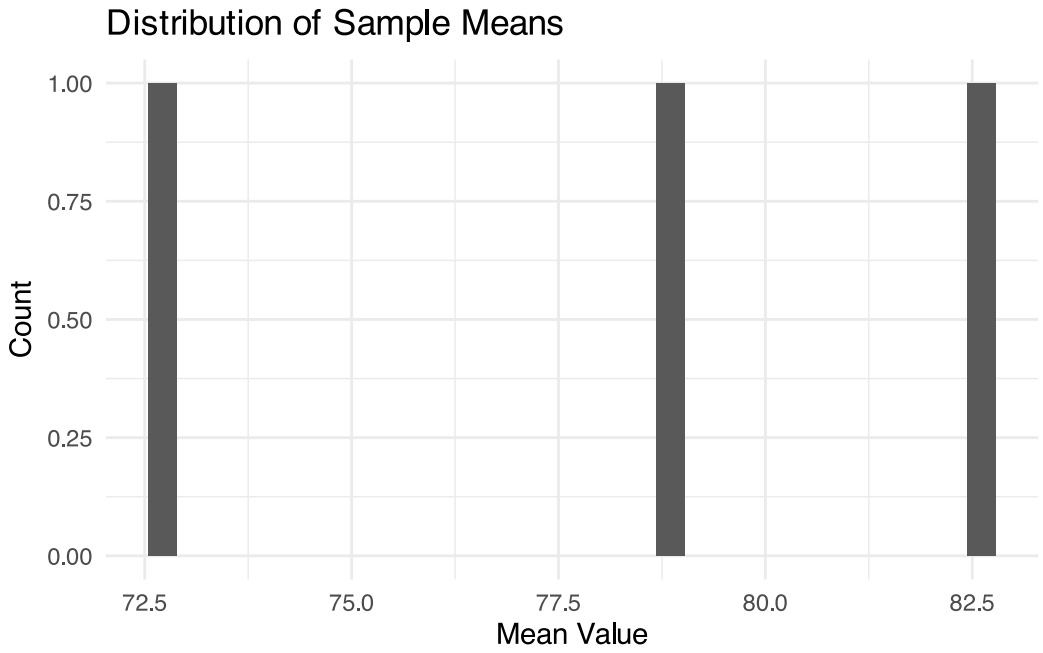
```r
sample_means <- tibble(mean_values = c(mean_1, mean_2, mean_3))

sample_means
```

```
# A tibble: 3 × 1
  mean_values
        <dbl>
1        72.8
2        78.8
3        82.7
```

See how Quarto automatically displays the table? In our `.R` file, we had to specifically print the table by typing `sample_means` on a line by itself. Here, Quarto shows us the output of our code automatically.

Finally, let's create a histogram of our sample means:

```r
ggplot(sample_means, mapping = aes(x = mean_values)) +
  geom_histogram() +
  theme_minimal() +
  labs(
    title = "Distribution of Sample Means",
    x = "Mean Value",
    y = "Count"
  )
```



The plot appears right after the code that created it. This makes it easy to discuss what we see in the plot right afterwards.

Why Quarto Makes Life Easier

You might be wondering why we'd use Quarto instead of just writing comments in our `.R` file. Here are a few reasons:

1. Everything is in one place - your explanations, code, and the results all flow together naturally.
2. You can write proper explanations using text formatting - headings, bullet points, even links to other resources.
3. The output (like tables and plots) appears right where you need it, making it easier to refer to specific results.
4. When you share your analysis with others, they can see both your code AND your thinking.

Workflow in Quarto

Working in Quarto is very similar to working with `.R` files, but with some powerful extra features. Let's look at how to work with a Quarto document:

Running Code

When you're writing a Quarto document, you can:

- Click the "Run" button (play icon) at the top of any code chunk to run just that chunk
- Use the same keyboard shortcuts we learned for `.R` files
- See the output right below each chunk as you go

Previewing and Rendering

One of the great things about Quarto is that you can create different types of documents from the same source. When you click "Render" at the top of the document, Quarto will create a final version that combines your code, text, and results.

By default, Quarto creates an HTML document that you can open in any web browser. This is great for sharing your analysis because anyone can view it, even if they don't have R installed. The preview will appear right inside RStudio, making it easy to see how your final document will look.

You can also render to other formats like PDF or Word documents. This is really useful when you need to share your analysis in different ways - maybe your professor wants a PDF for an assignment, or your colleague prefers to read documents in Word.

To preview your document as you work:

1. Click the "Render" button (or press Cmd/Ctrl + Shift + K)
2. RStudio will show you a preview of your document
3. The preview updates automatically when you render again

This makes it easy to build your document step by step and see exactly how it will look to others.

Hint: Just like with `.R` files, build your analysis step by step. Run each chunk as you write it to make sure it works before moving on. This makes it much easier to find and fix any problems!

This workflow lets you develop your analysis gradually, explaining each step as you go. It's like having a lab notebook where you can record both what you did and why you did it.

In the next section, we'll use these Quarto features to do a more complete analysis of some the AI Jobs data! We'll look at salary data from the AI job market and see how we can use sampling to understand salary variations across different job roles. This will show you how the sampling concepts we've learned can help us answer interesting questions about real data.

Analyzing Salaries in the AI Job Market

In our previous exercises, we practiced taking samples and calculating means using some simple measurements. Now let's apply these sampling concepts to the actual dataset and automate the sampling, rather than manually inputting each sample. We'll look at salary data from the AI job market and use sampling to understand the variation in salaries across different job roles.

The previous pages were about introducing you to how Quarto works - here we'll show what an actual analysis document might look like.

You can look at the actual `.qmd` file that generated this page here: `sampling-exercise-extended.qmd`

Setting Up Our Analysis

Load packages:

```
library(tidyverse)
```

Load Data

Now let's load our AI jobs dataset. This dataset contains information about various jobs in the AI industry, including salaries, job titles, and other interesting information:

```
ai_jobs <- read_csv("data/ai_jobs.csv")

# Let's take a look at what job titles we have
ai_jobs |>
  count(job_title) |>
  arrange(desc(n))
```

```
# A tibble: 10 × 2
   job_title                 n
   <chr>                 <int>
 1 Data Scientist           62
 2 HR Manager               57
 3 Cybersecurity Analyst    55
```

```
 4 UX Designer              54
 5 AI Researcher            51
 6 Sales Manager            49
 7 Marketing Specialist     48
 8 Operations Manager       44
 9 Software Engineer        41
10 Product Manager          39
```

Looking at the output above, we can see we have several different job titles in our dataset. For this analysis, let's focus on comparing salaries between different levels of automation risk. Our goal is to see whether there is a relationship between salary levels and the risk of a job being automated.

Hint: When working with real data, it's good practice to look at your data first before diving into analysis. This helps you understand what you're working with and spot any potential issues.

Taking Samples from Our Population

Instead of manually recording samples like we did in class, we can use R to take random samples from our dataset. Let's take 1000 samples of size 50 from the low and high automation risk groups.

When looking at this code, try to break it down into its component parts. We are using a `for` loop, which allows us to repeat a process multiple times. Start by understanding what is happening within the for loop - what are we doing each time?

- We're using the pipe operate to filter the dataset, then sample it, calculate the mean of the sample, and output the result into a list.

By repeating this in a `for` loop, we repeat this process many times, each time adding the new calculated sample mean on to the end of the list.

Next, understand how we tell the `for` loop how many times to perform the sampling.

- We define an `n_samples` variable at the beginning. Then, when starting up the `for` loop, we tell it to loop from 1 to `n_samples`: `for(1:n_samples) {}`.

```r
sample_size <- 50
n_samples <- 1000

# Set up an empty list to add to
high_sample_means <- c()

for (i in 1:n_samples) {
  # Sample the High automation risk observations
  mean <- ai_jobs |>
    filter(automation_risk == "High") |>
    sample_n(sample_size) |>
    summarise(mean(salary_usd)) |>
    pull()
```

```
  # Add to the sample_means list
  high_sample_means <- append(high_sample_means, mean)
}

# Repeat for the Low risk group
low_sample_means <- c()
for (i in 1:n_samples) {
  # Sample the High automation risk observations
  mean <- ai_jobs |>
    filter(automation_risk == "Low") |>
    sample_n(sample_size) |>
    summarise(mean(salary_usd)) |>
    pull()

  # Add to the sample_means list
  low_sample_means <- append(low_sample_means, mean)
}
```

Now we have our sample means! Let's combine these into a single table:

```
means_table <- tibble(
  "High" = high_sample_means,
  "Low" = low_sample_means,
)
means_table
```

```
# A tibble: 1,000 × 2
     High     Low
    <dbl>   <dbl>
 1 80599.  96332.
 2 79619. 100086.
 3 82803.  96272.
 4 80348.  95943.
 5 83480.  98276.
 6 84768.  98346.
 7 80532.  97289.
 8 87658.  94836.
 9 77582.  92619.
10 81128.  94673.
# ℹ 990 more rows
```

And reshape it into a tidy long table format. This just means we're stacking the two columns from the table above into one column with a label for which automation risk the sample is from:

```
means_table <- means_table |>
  gather(key = "automation_risk", value = "sample_mean")
```

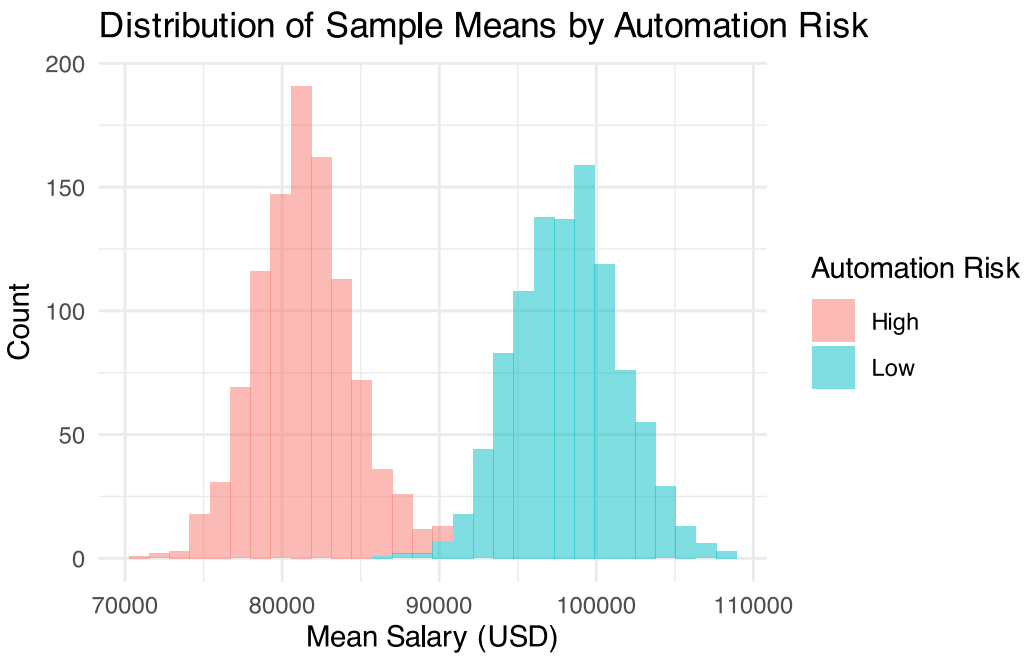
```
means_table
```

```
# A tibble: 2,000 × 2
   automation_risk sample_mean
   <chr>                 <dbl>
 1 High                 80599.
 2 High                 79619.
 3 High                 82803.
 4 High                 80348.
 5 High                 83480.
 6 High                 84768.
 7 High                 80532.
 8 High                 87658.
 9 High                 77582.
10 High                 81128.
# ℹ 1,990 more rows
```

Visualizing Our Sample Means

Now we can create a visualization to compare the distribution of sample means between these two job titles.

```
# Create a plot comparing the distributions
ggplot(means_table, aes(x = sample_mean, fill = automation_risk)) +
  geom_histogram(alpha = 0.5) +
  theme_minimal() +
  labs(
    title = "Distribution of Sample Means by Automation Risk",
    x = "Mean Salary (USD)",
    y = "Count",
    fill = "Automation Risk"
  )
```



Connecting to the t-test

This visualization shows us how the sample means are distributed for each job title. The overlapping histograms make it easy to compare the two distributions. Another way we could visualise this is with side by side points with error bars:

```
# Create a plot comparing the distributions

# We need to start by calculating the mean and
# standard deviation of the sampling distributions
# to pass to the errorbar layer:
summary_stats <- means_table |>
  group_by(automation_risk) |>
  summarise(
    mean = mean(sample_mean),
    sd = sd(sample_mean)
  ) |>
  ggplot(aes(y = mean, x = automation_risk, color = automation_risk)) +
  geom_point() +
  geom_errorbar(aes(ymin = mean - (2 * sd), ymax = mean + (2 * sd))) +
  theme_minimal() +
  labs(
    title = "Distribution of Sample Means by Automation Risk",
    x = "Automation Risk",
  ) +
  scale_y_continuous(name = "Mean Salary (USD)", labels = scales::comma)
```

This plot shows us quite clearly that the sampling distributions are quite different between the different automation risk groups.

Think back to our Hypothesis Testing using the t-test. The goal there was to tell whether the difference between the means in the two groups was significantly different. How we define statistically significant is based on the estimated sampling distribution. We estimated this based on just a single sample. Here, we're directly looking at the the actually sampling distributions (for sample size = 25). Another way to interpret the t-test statistical significance is to look at whether these sampling distributions are far enough apart *and have a low enough variance* that their standard error bars don't overlap.

In the boxplot, we can see that the external lines of the boxplots (representing 2 standard error) don't overlap with each other. This would indicate that, with at least 95% confidence (remember, 2 SE encompasses 95% of the distribution), the mean of a given sample of 50 salaries from the High risk group would not overlap with the mean from the Low risk group.

Let's translate this into a t-test:

```
# Start by drawing a new sample of 50 from each group
high_sample <- ai_jobs |>
  filter(automation_risk == "High") |>
  sample_n(sample_size) |>
  pull(salary_usd)

low_sample <- ai_jobs |>
  filter(automation_risk == "Low") |>
  sample_n(sample_size) |>
  pull(salary_usd)

# Run the t-test
t.test(high_sample, low_sample)
```

```
    Welch Two Sample t-test

data:  high_sample and low_sample
t = -2.4373, df = 93.883, p-value = 0.01668
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -23712.004  -2421.726
sample estimates:
mean of x mean of y
  88295.8  101362.7
```

Yes! Our t-test results match up to what our plots showed! This demonstrates how the Hypothesis Testing framework allows us to estimate patterns in the sampling distribution even from a single sample.

What Have We Learned?

This exercise shows how sampling helps us understand patterns in real-world data. Instead of looking at every single salary in our dataset, we can take samples and use their means to get a good idea of typical salaries in different groups.

Some key points to notice:

1. We used the same basic sampling concepts as in our class exercise, used the power of R to take many simulated samples.
2. We can easily compare different groups (job titles) by taking samples from each group.
3. Visualizing our sample means helps us see patterns that might not be obvious just looking at numbers.
4. We looked at the relationship between simulating the sampling distribution to estimating the properties of the sampling distribution to apply in hypothesis testing.

Bibliography