SEL2229 Exercises PDF

2025-01-28

Contents

Week 1: New Tools

Aims

Set up accounts with new tools, become familiar with Posit, understand how to complete weekly reflections in Posit for your first assignment.

Introduction

Before we get started, let's understand a bit about how these labs will work. Each week in lab is a chapter in this "book", associated with a checklist as a separate document, and reflection questions that you will find in an assignment template (more on this soon). Because you will be working through this independently at your own pace, there is a fair bit of text to guide you; where you see a colon before a :word or phrase, you can click on it to unpack some extra information or context. Note: this will not work within this PDF version. You must use the online version to enable this feature.

The entire guide is written in R and RMarkdown using bookdown (this will mean something to you shortly): you can navigate through the whole thing using the table of contents on the left, and use the menu at the top to show/hide the table of contents, search all the text, change how it looks while you're reading, or even download the whole thing to read offline. I've also provided a checklist of all the tasks that you can use to actually tick things off. **Anything marked as "Bonus" is not part** of the assignment, so it will not appear in the checklist; but it will be useful to you in learning stuff.

You might be able to work through some steps without looking at the guidance, especially as you learn more and more about how to do things. But do this at your own risk - **you need to make sure you're actually meeting the full requirements of the checklist, because this is part of what you will be marked on for your first assignment**.

While you're working your way through the checklist, you might get stuck - this is part of the learning process. The instructions I provide here are general and meant to guide you through figuring things out for yourself. From time to time, we may deliberately put bugs into our lab tasks to help you develop this skill. The good news is that when you do get stuck, we're here to help - just raise your hand or call me over and someone can help you figure it out. You can really ask us anything. You might also want to chat to your peers nearby - if you've had a problem, someone else has probably encountered it, too - they may have already found a solution they can share. Overall, especially when it comes to code, what's important is that it works and you can explain how it works - but not every problem has a single solution!

If you run into something you want to ask about after lab - since you may need to dedicate time to finishing a week's task outside of the timetabled hour - use the dedicated Discussion board to raise a question on Canvas. I'll hop in as soon as I can to help.

Use the central checklist on the left to jump to particular tasks for the week as needed - but note that the walkthrough is meant to be... walked through. You should read it all and work through it sequentially. To keep overall track of your progress in each week coming up to the assignment deadline, use the full checklist document.

Although the tasks aren't numbered, they're designed to be done in the order they're listed/presented here. In many cases, a single task will involve a few steps that I will outline in detail here.

Perusall

Context

Perusall is a tool for collaborative reading. If you're like most students, you do your reading on your own. Perusall allows you to get more out of the reading by interacting with others over the text, much in the way

you might comment on a post or article elsewhere on the internet. "Others" here means other students on the module, but also me as the module leader - each reading will be released with my annotations, which should help you to understand exactly what I expect you to get out of the reading, often with explicit ties to other parts of the module (lectures, labs, and assignments). If you comment on a reading (even if this is anonymous), I'll try to answer.

This may come as a shock, but there is a point to this beyond it being cute and fun: getting in the habit of engaging in the reading this way means you are more likely to actually keep up with doing it (which, of course, you were planning to do anyway, right?), and as a consequence, you are more likely to do well in the module.

Tasks

- **Sign up**: Go to the Perusall website and create an account using your university email.
- **Join module**: Perusall is a service that anyone can use, and there are loads of modules on it (referred to as "courses" to cater to the US market). Figure out how to use the join code CUSKLEY-B2XB3 to access readings for SEL2229.

Now explore around a bit. Find the reading for week 1 and explore it a bit; it's already annotated. Try out some features:

- Create or reply to a comment
- Search the reading for all instances of the word 'variable'
- Figure out how to upvote existing comment/annotation
- Explore the hashtags for the module
- Get Perusall to read aloud to you

You'll need one of the phrases I've annotated for your reflections shortly, so keep the tab open.

Posit

Context

Posit is an *interactive development environment*, also known as an IDE. In other words, it's a piece of software that you can write (i.e., develop) :programs or code in (in this case, in the programming language known as R). It's important to note that Posit used to be called RStudio, and the in-browser version used to be called called RStudio Cloud.

You'll need to know this for troubleshooting; this is just a fancy word for figuring things out if you have a problem. Like any other software, you might Google what you want to know with the name of the software (e.g. "how do I make a table in Word"). Because they've renamed RStudio to Posit just in the last couple years, searching something like "how do I create a new script in Posit" may not get you the solutions you need, so be prepared to search "how do I create a new script in RStudio" instead. **Also note that Google is ruined** - the default is now that you need to scroll past AI generated garbage results telling you to eat glass to get to answers that will actually be useful to you. **You can add "-ai" to any search to suppress this, e.g., search "how do I create a new script in RStudio -ai".**

You will need to do this - troubleshoot problems using Google - as you're learning R with Posit at some point. In fact, you will probably watch me do this to help find answers to problems during labs. This is not just a performance for teaching purposes: I use R/Posit all the time, and have for a decade, and I am constantly searching for solutions, often using Google.

Your aim today is just to get signed up to Posit, look around a bit, and set up the project that you'll eventually submit for your first assignment. You're not expected to understand much of what you see immediately; don't let it scare you, just poke around. Next week we'll start on our journey of actually understanding how to use it; by having a look now, you'll make that start a little bit easier on yourself.

Tasks

- ☐ **Sign up**: Go to the Posit website and create an account. Once you're logged in, you want to go to Posit Cloud.

Once you've done this, it will stick you somewhere called "Your Workspace". Throughout the module, you'll do your seminar work in an R Project. Think of an R Project like a bundle of files related to your work that are all in the same place so they can talk to each other. You'll submit this bundle for your first assignment. We'll work on a range of exercises, some adapted from Bodo Winter's Statistics for Linguists: An Introduction Using R, which will also be the target reading for the first few weeks. Eventually we'll get into more complex things, and you may need to port files from that into your R Project. But first, let's get it going.

- **Access the template** using this link. You will use this as a template for your first assignment, working in it over the next 7-8 weeks. Quickly look around and see what you can make sense of. Where can you type? Where can you click? What are the options in the different menus?
- At the top of the assignment template, there's a red warning that says 'Temporary Copy'. If you think about it even *very* briefly, you probably do not want the project you're using to prepare your first assignment to be temporary. Temporary means it will blink out of existence as soon as you close the browser window. Figure out how to make this a permanent copy.
- Why might it be that any project shared in this way defaults to a temporary copy? :hint

Once you've gotten your assignment template saved as a project within your own account, think about what you're looking at. There are four panes in Posit, and each has several tabs. For now, we'll only focus on the default tabs.

- The **top left pane** displays files (which you can edit) or variables/data (which you can only look at).
- The **bottom left pane** displays the console. This is where you interact directly with R, or where lines of code are sent to execute or run.
- The **top right pane** displays the environment. This shows what variables and objects are loaded into your R session.
- The **bottom right pane** shows files - this is where your reflections file is (`assignment_reflections.Rmd`), and where you'll put anything else you work on in Posit throughout the semester.

Start by going to the files pane and opening the file `assignment_reflections.rmd`. The next thing you're going to do is switch into Posit and work your way through this file, which has specific questions for you to answer (some of which you'll be able to do right away). But before we do that, let me explain a little bit about this `.Rmd` file, and files in general:

- The *file extension* here is `.Rmd`, which you probably haven't seen before. Extensions you have probably seen are things like `.docx` (Word), `.xlsx` (Excel) and `.txt` (plain text). Note that because file extensions start with a ".", filenames can't include one (for any kind of file).
- `.Rmd` stands for R Markdown. You can read more about markdown in general here, but the gist is it's just a way of creating text that can have formatting (e.g., bold, italics) much like Word.
- Ok great. So why can't you just do this in word, Which you've almost certainly used before? A few reasons:
  - Part of the point of this module is to learn to use R/Posit, not Word.
  - RMarkdown can do things Word can't, like embed code. Look at the beginning of the assignment reflections file and you'll see a code chunk (the file itself explains a bit more).

It used to be that using RMarkdown involved *really* learning markdown itself. This isn't really hard, it's just different: for example, instead of selecting a paragraph and then selecting Header 1 from a menu, you

would just put a single hashtag in front of the paragraph (and two hashtags is Header 2, etc). Learning RMarkdown in this way is still useful (if you type fast, it makes everything faster because you're not fiddling with point and click mouse stuff to format text). However, Posit now has a "visual" mode for RMarkdown that just makes it into a simplified little word processor that should be very easy to use (and if you want, you can still use normal markdown syntax).

The default for an RMarkdown file in Posit is "Source" mode, which shows the hashtags etc. that it would typeset into e.g., headers. **On the menu within the file's tab, click "Visual" to see this word processor style instead.** It may prompt you to install some R packages required to make this work (e.g., Pandoc) - go ahead and accept this suggestion (we'll get to it later, but think of a package like an add-on).

Now you're ready to get on with your reflections for this week. The document itself has more information about what these should look like - **pay close attention to this**. I strongly suggest you complete these reflections now when things are fresh in your mind, and you can rest easy knowing you've already started your first assignment! This recommendation holds for every week: **you should do your reflections during or immediately after completing the lab task**. Trust me when I say that *you cannot do this all at once right before the deadline*; all the extensions in the world would not make this more feasible either. The lab work and reflections are designed to be done *as you go* throughout the module, and is not amenable to cramming.

Bonus: Zotero

Find bonus content on how to use reference management software here.

Week 2: Introducing R

Aims: Become familiar with Posit and basic ways of working with R.

Getting started with R

This week's lecture and reading have given you a more detailed introduction to R and statistics, and using R within RStudio/Posit. In today's lab, we'll start with :some exercises to explore Posit, in addition to adapted versions of some of the exercises at the end of Winter (2020), Chapter 1.

As you should be aware, I do expect you actually do the reading, and you should be doing it ideally ahead of lectures and labs, or at least looking at it before this point I may refer to specific parts of the reading in lab, and you may want to have the reading open in another window or tab to help you as you go.

For the most part, R will tell you when things aren't working out by throwing an error; this will be some information (usually in red) printed within the console. If you're stuck, just flag me down or look for hints. Also remember that there is an easy to use search function within Perusall; you can use this to find exactly where in the reading Winter talks about a particular task or concept.

Before you begin, here are few hot tips to remember:

- As we learned last week, within a script or RMarkdown file, writing code and running it are two separate things. If you enter code directly into the console it runs immediately.
- Code is **case sensitive** - that means that `x` and `X` are two different variables.
- There is a difference between a variable name, e.g., `my_variable`, and a string, which occurs inside quotes e.g., "my variable". Strings can have spaces in them, but variable names cannot. This is why you will often see underscores (as in `my_variable`), or camelCase, e.g., `myVariable`. Note that because R is case sensitive, if you define something as `myVariable` and then type myvariable, it will throw an error.
- **The vast majority of errors are caused by typos**, especially when you're starting out, so check that carefully first. R doesn't spell check - it assumes you mean what you type (think about why this might be in reference to the prior point). - Certain words "reserved" - this means that you can't use them for variable names because R already uses them for something. For example, we'll use the `summary()` function below - the fact that this already exists means you can't create your own thing called "summary".
- The same goes for many characters, e.g., you can't call a variable `x+y` because R will try to interpret this as trying to add the variables `x` and `y` together.
- There are some useful keyboard shortcuts you can use, especially when you're working in the console:
  - Pressing the up arrow will "scroll" through what you've typed before, so if you need to re-execute a bit of code you typed earlier (e.g., because the value of a variable has changed), just arrow up to find it and press enter.
  - If you start typing in the console, R will pop up a list of what it thinks you might be looking for, and you can use tab to select from the top of that list (and arrow down etc).

Exploring the interface

We'll start this week where we left off with Posit last week: having a look around and getting to know things better. As you'll have seen in my notes on Perusall for this week's reading, there are a few differences between what Winter (2020) discusses and what we have to worry about because we're using a cloud setup.

- Open the project you saved from last week, your personal copy of the assessment template. It will take a minute to "wake up" - why might this be? :answer

- **Explore the view:** Try hiding panes you can see and showing ones you can't; you can also change their sizes, or the tabs you're viewing within a pane. Why might you want to show or hide a pane?
- **Do some math:** Find the console and calculate the square root of 49,305. Then, assign this to a variable and multiply it by itself to :check it's working like you would expect. :hint
- **A simple script:** Create a new R script that includes the code in the previous step for finding and checking the square root of 49,305 *and* checking it. Also ad code to the script that assigns a vector of 10 numbers to a variable called `myNumbers`, calculates the square root of each number, and assigns that to a new variable called `mySqRoots`. Once you've made the script, run it and make sure it goes through without errors. :hint
- **Binomial test:** My 13yo son was born in Edinburgh, but he moved to Italy when he was 1 and didn't move back until he was 5. He has a weird US American accent, probably because he watches too much YouTube. But I was also curious about his lexicon, so I showed him a picture of this vegetable 50 times and asked him to name it. He called it a "courgette" 37 times. Do a binomial test to see if his lexicon is leaning British or American. :note

Dealing with external data

Much of what R is actually used for is taking data and doing something with that data, be that calculations, summaries, or analyses. As such, it's incredibly important to understand how data gets *into* R, and how we can look at it from there.

- **Upload a file:** In the reading, you'll have noticed Winter references "a folder where the files from this book are" (he actually tells you where to get this in the Preface). All the materilas from the book are in an OSF repository, which you can find here (OSF stands for *Open Science Framework*).
  - Find the file `nettle_1999_climate.csv` and download it. :hint
  - Upload the file into Posit. :hint

Now that you've gotten this far, the next steps are all embedded in your weekly reflections; note that it will ask you to reproduce some of the code inline (in code chunks). **Make sure the code in those chunks actually works**. Here's a brief description of what you'll need to do:

- Read the file into a variable called `nettle` :hint
- Look at the first six rows and the last six rows of the file.
- What kinds of variables are in the dataset? Use the `summary()` function to find out a bit more about it.

Bonus: Swirl

Swirl is an interactive tool for learning R that works within R itself. In the reading I gave some context for what an R package is - Swirl is just a package that lets you learn how to use R in basic (and more advanced) ways.

Swirl is a bonus here because you can use it independently; it really is meant to guide you along step by step. I probably need more hobbies, but I might even call it fun. For now, I'm only recommending the first few modules in Swirl, but if you're wanting to practice or advance in R, you might continue to work throught Swirl on your own.

While you use it, I want to add a layer for you: how do you think this *works*? Think about what Swirl is doing to interact with you - we've come a long way, but even just a couple decades ago people wouldn't have hesitated to call this AI. I haven't actually looked at the R code that makes Swirl work, but I'm betting it relies extensively on the kinds of logical operations Winter (2020) discusses in 1.7, Ch.1.

- **Start Swirl:** Follow the instructions in section 1.17.2 of Winter (2020) ch.1 to get started with Swirl
- **Completing modules:** The remaining tasks on the list for Swirl are modules within the program itself. It will do a good job of instructing you on how to select each one.

Week 3: Scripts & Troubleshooting

Aims: Get used using scripts, and start to understand tidying data, including wrangling, cleaning, and re-coding.

This week in lecture we started to look at some actual experiments, and the kinds of data they generate. While the reading in Winter talked a fair bit about *how* we might restructure and tidy data, our lectures focused more on *why* you need to do that with real data. Now, you're going to spend some time actually getting that done with real data yourself.

As you go through this, **remember**: there is more than one way to solve these tasks. Recall last week, even with very simple problems (e.g., generate a vector of ten numbers), there were :multiple ways to approach a given task. It doesn't matter all that much how you do it, just that you get the outcome you need and you understand how you got there. Like learning a natural language, learning a programming language is difficult - but think of this functionally. Most people don't try to sit down and learn the entire grammar of a natural language by rote; they start by focusing on how to order a coffee and say good morning. As you become more advanced in your R skills, your code will get more tidy, and your understanding of the "grammar" of R will come, but for now, worry mainly about using your code to solve problems - efficiency will come later.

Using documentation

Below, I've provided a list of functions this week's exercise is using with links to their documentation. It's a good idea to start by poking around the documentation for each function to understand a bit more about what it does and how it does it, including the kids of arguments each function expects and whether there are any relevant defaults.

- read_csv()
- filter()
- mutate()
- rowwise()
- ungroup()
- select()
- pivot_longer()
- View()

Note that there are also lots of R cheatsheets that you can download for reference as PDFs, or just bookmark for later. There is one for tidyr and one for the RStudio (Posit) IDE. For the RStudio one, just keep in mind that there might be minor differences since you're using Posit online in the browser instead of locally.

Troubleshooting

You're going to make mistakes when you write code, so being able to spot them and fix them is a key skill. The nice thing about R, relative to a point and click program like Excel or Google Sheets, is that it will usually tell you when you make an error. More often than not, errors in your code will be incredibly stupid: things like misspelling a variable name, forgetting to add a pipe, or load a file. This exercise involves finding those kinds of errors and fixing them.

Note that this is a domain where tidy pipes might hurt us a bit; every bit of code "piped" together is run as one, meaning we can't run long pipes line by line to troubleshoot exactly where an error is coming from in our code. However: tidyverse will try to tell us where our pipe is broken, so pay attention to the errors it generates.

Using an existing script

- **Get the files:** To do this troubleshooting exercise, you'll need to download some files from another project and upload them into your posit project. Download the files `colourTidy.R`, and `colourData.csv` from this project. Note that `colourData.csv` is inside a top level `Data` folder.
- **Organise and Upload:** You're putting a lot of work from different weeks into this project, and it might get messy if you're not careful. You need to organise your project so I can make sense of it when it's submitted - when I'm marking, I'll be looking closely at your reflections file, but *also* your scripts and whether you have all the right data files in the right place.
  - Create two new folders: one called Data and one called Scripts
  - Move your scripts from earlier weeks into the scripts folder, and the `nettle.csv` data file and the `Week6_AJT_AEP.csv` file (which we'll work with in week 6) to your Data folder.
  - Upload the `colourTidy.R` script to the scripts folder, and `colourData.csv` to the Data folder.

Fixing bugs

Now it's time to dig in and fix this script. **It's important for the purposes of your assignment, but also to get used to commenting scripts, to use comments to explain what you find while you're debugging.** In addition to this, when you're working with a script you didn't write, it's always best to preserve the original script so you can check back to it to see if you accidentally introduced something you didn't intend to. Start by creating a copy of `colourTidy.R` called `colourTidyR_Troubleshooting.R` that you'll use to fix the bugs in the original script. You can do this by creating a new, blank script and copying and pasting everything in; but you can also select the file in the files pane and use the gear icon to make a copy of it.

For each issue you find and fix, create a corresponding comment that explains the problem and how you have solved it. I'll tell you the first few things you're likely to encounter for free:

- You may not have the tidyverse package installed - because that's what the script uses, it will immediately fail unless you do install it. As soon as you open `colourTidy.R`, Posit can tell and it will prompt you to install this - go ahead and do that.
  - **Note that because most every script you use will use the tidyverse, and because you want your scripts to be able to standalone, the first line of *all* your `.R` scripts going forward should be `library(tidyverse)`**
- You've started your own way of organising scripts and files, so on the second line of the script where it tries to load the data file, it won't be able to find it. You need to change the filepath, i.e., the argument string passed to `read_csv()`, so that it makes sense *relative to where your script file is*. That means you need to use the path to tell it to go *out* of the Scripts folder into the main level *and then* into the Data folder where you put the file. To get you started, you use "../" to get out of the folder you're in and "up to the next level.
- If you're still getting a problem with the script loading the data, look *very* carefully at the filename.

Adding comments

Once you've fixed all the bugs and everything works, it's time to add some comments. Comments start with a `#`, and represent information about the code, but the `#` at the start of the line lets R know that whatever

comes after it shouldn't be run. This is primarily used to explain what code does for others or your future self, but can also be used for "debugging": you can comment out a specific line of code to find out if it's causing a problem, or to test something without that line. See the example below:

```
#Define x
x<-6+3
#Define y
y<-9*3
#Define z as y divided by x
z<-y/x
#will test the square root of z later
#myvar<-sqrt(z)
```

- Add comments to `colourTidy.R` throughout, indicating what each type of command is doing to the data. Note that this need not be every single line, since some lines repeat the same kind of operation

Week 4: Tidying & Visualisation

Last week, you fixed my sloppy scripting. This week, we'll build on those skills to write your own cleaning, coding, and wrangling code, before pivoting to an introduction to visualization. Throughout, remember to keep your troubleshooting hat on: push through mistakes if you make them as you go, and remember to call me over if you get stuck, or open a discussion item on Canvas if you're finishing the exercise after lab. Chances are I will just help you Google something - but I'm here to support you.

Tidying

Cleaning

Start with the tidy dataframe - `tidyDat` - from last week. This is tid*ier* than the raw data you started with, but there are still some imperfections. Using the `tail()` function (type `?tail()` in the console if you're unsure how to use this, or you can google it), look at the last few rows of the data frame. :hint

- To focus on this week's work, start by creating a copy of your finished `colourTidy_Troubleshooting.R` called `colourTidy_Week4.R` in your scripts folder.
- Having looked at the last bit of the data frame, what needs to be done? Using the appropriate tidyr function, clean the data. You can either pipe this into the existing tidy commands, or make new ones - it's up to you. Just make sure it does what you want it to do!
- Once it works, add a comment to the file to explain what you've done to the data and why. The other code should already be commented from last week, and you should **plan to add comments to code you add to the file to show that you understand what the code is doing**. I won't include this in the instructions explicitly again, so build this in as a habit.

Coding

For both of the tasks below, you will want to consider the `str_detect()` function. Start by googling this or using `?str_detect()` in the console to look at the documentation.

- In the console, use the `View()` function to visually scan he values of the `ArtHobbies` variable. Consider whether these values are useful as-is.
- In the script, using the `mutate()` function, add a column called `isArtist` that makes participants responses to `ArtHobbies` more useful. You can do this using values like "Yes" and "No" or 1 and 0.:hint
- One of the main things this student was interested in about the colour terms participants provided is whether they were "simple" or "compound". Simple colour terms are one word (like *lavender*), while compound ones are two or more words (like *light purple* or *blue-green*). Think about what kinds of characters compound colour terms contain that simple ones don't. In the script, use the `mutate()` function to add a column called `isCompound` that codes whether a term is simple or compound, again using "Yes/No" or 1/0. [:help] (#Help)

Wrangling

In class - and in the code you just fixed in the troubleshooting section - we looked at moving things from wide to long on a pretty big scale. But wrangling can also mean creating valuable summaries. In this final step,

- Create a new tibble called `compoundProportions` that will show the proportion of compound colour terms provided for each colour (i.e., the number of compounds divided by the total) for artists and non-artists.

- Use `group_by()` to to group by isArtist and ColourHex, and the `summarize()` function to calculate the relevant proportion in a column called `compoundProp`.:hint

Introduction to visualization

Data visualisation

Data visualisation - and visualisation in general - is a powerful tool for communication and organising both data and ideas. In general, you're probably used to focusing on writing things in your degree. Clear and effective writing is extremely important, but sometimes, as the saying goes, a picture is worth a thousand words.

The tidyverse does data visualisation using a package galled ggplot2 - gg stands for *grammar of graphics*. The whole concept here is that there is a generative structure to how you build graphs (not unlike generative grammar in language), and it can be additive: you can layer parts of a graph on top of each other to make very complex visualisations.

There is literally an entire book on this package (probably more than one), and the shortest introductory video I could find on YouTube is 26 minutes (and it's probably bad) - so we are not going to learn this in a day. We'll continue working with ggplot in weeks 6 - 8 (next week, we'll take a short break from the tidyverse and focus on survey software).

To start to understand ggplot, you'll play a bit with the same data we've already been using to understand how it works. Before doing that, read sections 1.1 and 1.2 of the ggplot2 book if you haven't already (ignore the warning that says not to read it; I've vetted this bit). This will give you some background and help you to make sense of what you're doing.

Understanding plotting code

To start, you're not quite going to be making your own plot from scratch - you'll begin by looking at how I've made a plot and fiddling with my code to understand how ggplot works.

- Start by adding the code below to your `colourTidy_Week4.R` script at the bottom (copy and paste it in). Add a sensible comment just above where you paste it in.

```
hexvals<-unique(tidyDat$ColourHex)

#defines an object for the plot, using data compoundProportions, setting isArtist to the x axis, and pr
myplot<-ggplot(data=compoundProportions, aes(x=as.factor(isArtist),y=compoundProp, colour=ColourHex,grou
  geom_point()+
  geom_line()+
  ylab("Proportion of Compound Colour Terms")+
  ylim(0,1)+
  scale_x_discrete("Is Artistic?",labels=c("Not Artistic","Artistic"))+
  scale_colour_manual(values=hexvals,guide=F)+
  theme_bw()

# show the plot in the lower right pane
show(myplot)

# save the plot to a file
ggsave("myTidyColourPlot.png", width=8, height=8)
```

- Run the whole script and see what you get (you should see a plot displayed in the 'Plots' tab of the lower right pane).
- Now, explore and figure out what each line of the ggplot call does.
  - Comment out each line below the first line of the plotting call (`myplot<-ggplot(...)`), and re-run (Source) the script each time.
  - Look at the plot appearing in the Plots display, and make note of how it changes depending on what you comment out. Use this to figure out what each bit of code is doing, and add a comment above each line of the plot call to explain what that line does

Explore layers and geoms

ggplot has lots of different kinds of geoms that you can layer on top of each other; a line geom uses data to draw a line, point geom uses data to draw points, etc. Briefly explore the different kinds of geoms here.

- What happens if you add `geom_bar()`?
- What happens if you add `geom_boxplot()`?

Week 5: Creating surveys

What is a survey?

The word "survey" contains multitudes; in many fields, this is a very specific method for gathering data involving particular kinds of questions. However, for our purposes, think of this as a specific kind of tool: a survey is a way to create a computer-based series of questions that you can send to participants. You might think this is very straightforward; just make up some questions and throw them in. But it is rarely this simple: you have to think carefully about how the way you ask questions can constrain participants' answers, what kind of answer you really want from them (e.g., a word? a choice? 1-2 choices? a number?), and whether some questions are only relevant for some participants. In short, a survey is more complicated than just a list of questions. If an acceptability judgement task makes up a big part of your second assignment, you'll get up close and personal with these kinds of details as you design your own study.

This exercise will focus on a software that allows you to create surveys that can be distributed online. There are a lot of options in this space, but many of them cost a lot of money, and/or the software provider can see or keep the data you collect from your participants.

Creating a survey

This week, you'll spend most of your time outside of Posit and in a tool called Survey.js instead. Under the hood, this is actually in a completely different programming language, JavaScript. However, don't panic: you aren't actually going to learn to write JavaScript, Survey.js provides a point and click interface that writes it for you.

- Go to surveysjs.io and create an account.
- Once you've done this, go to the My Surveys area of your account and click 'Create a Survey'

Now you can use their little editor in the browser to make quite a complex survey. We'll work through this step by step, but you have to make some decisions about the topic of your survey. Before you start, take a minute to think about something it would be interesting to look at in terms of acceptability and dialect. Think of something that may be standard in your own dialect (and maybe some related ones), but might be considered unacceptable by speakers of other varieties. :hint

- Use the menu on the left to add a question - start with a Radio Button Group that obtains informed consent from your participant. This should be on it's own on the first page; the rest of the questions below should be on Page 2. Make sure to give each page a reasonable name/description (it's not very polished for your participants to see "Page 2")
- Give the survey a title and a description; just make it something about language, e.g., "Acceptability study"
- Add a Single-Select matrix question that gives the participant at least two sentences related to the phenomenon in your own dialect. The sentences should be in the rows, and there should be five columns labeled "acceptable", "okay", "unsure", "odd", and "unacceptable".
- Using the Yes/No (Boolean) question type, create a separate categorical acceptability item for each sentence or item.
- Using the Dropdown question type, create a question that asks for the participants' age. You don't want an exact number so create relevant ranges.
- Create a Radio Button Group that asks where the person is from using a few relevant regions or areas, including where you/your dialect are from (e.g., this might be cities, or much larger regions like England, Scotland, etc.). Remember to include an "Other" option.
- Create a Single Line Input item so that participants not listed in your existing list of regions can say where they're from.

- Create another Single Line Input item for participants who select the area where you/your dialect are from asking how long they have lived there.

Now you should have a neat little survey. Use the "Preview" button to see how it looks for participants. Make sure everything looks like you want, there's no typos or unclear instructions, and the questions appear in a reasonable order.

Logic and validation

This looks great already, but there are a couple of problems:

- The participant can choose what to answer and what not to answer; we need to make most of the questions required.
- If the participant selects 'No' in the consent question, they can still do the survey.
- We don't really want someone who chooses a specific region to answer the open-ended question about where they are from.
- We don't really want the question about how long people have lived in the relevant region to appear unless they answer that they are from that region.
- Make everything but the two single line text questions required using the "required" button on each.
- Go through each question and give it a transparent Question name in the settings (on the right side) under the General menu (e.g., "consent", "Age", etc.). This will make it much easier to find as you add validation and logic in the next few steps.
- Likewise, under 'Choice options' (for all but the single line text questions), make sure the *value* of the choice matches the display text (note that for the Single-Select Matrix question, this will be under 'Columns' and 'Rows')
- Under the settings on the right side for the consent question, scroll down to Validation and click 'Add Rule'. Under error message, add "You must consent to participate. If you don't want to, simply close the window now. No data has been collected."
- Click the little magic wand next to 'Validation Expression', in the window that pops up, under 'select question', select the consent question from the dropdown menu and set it to 'Equals' Yes.
- Now test that this worked: If you preview your survey and select "No" and try to go to the second page, it should show you an error.
- Now, you want to make the single line text questions only visible for the relevant people.
  - In the settings for the open-ended question about where a participant is from, scroll to the Conditions section. Use the magic wand next to 'Make the question visible if' to select the region question, select Equals and 'other'.
  - In the settings for the open-ended question about how long someone has lived in the target region, go to the conditions section and click the magic wand. Do the same as for the previous question, only tie it where the selection equals the relevant region.

Before you move on, use the preview button to stress test your changes. Make sure the relevant questions are required, you can't proceed unless you can consent, and that the open-text questions only appear if the specific selections are made in previous questions.

Moving it to your assignment

We don't quite have time to *really* get your little survey online - that involves an extra step or two, including getting a persistent URL you could actually send to participants, and storing responses in a database. However, it's worth noting that for your third year project, if a survey is involved, building it in Survey.js will get you a long way: you can build it yourself and share it with your supervisor to polish and refine it using everything you've learned today (after that, you would just get in touch with me to get it online and start collecting data).

For today, you'll just get your proof-of-concept survey into your R project as part of your assignment portfolio. There's already a folder called `Week5_SurveyDemo`. This just has one file inside it: `index.html`.

- Open index.html in Posit to prepare to get your survey data into it from Surveys.js. You want to leave this file as-is, except you want to replace the part on line 21 that says `/*PASTE JSON FROM SURVEY JS HERE*/` with your custom JSON.
- Go to the JSON Editor tab in Surveys.js. If you look closely at what's here, it's all the information about your questions in a specific format called JSON (which stands for JavaScript Object Notation). However, don't worry much about reading through this; the point of the Design tool is that you don't have to write (or read) JSON.
- Select all the JSON here and copy it to the clipboard. Go back to the `index.html` file and paste it all in after the `const surveyJSON =`, replacing the `/*PASTE JSON FROM SURVEY JS HERE*/` text with your custom JSON.
- Save the index.html file. Now, in the files pane, click on the `index.html` file and select 'View in a web browser'. This should open your survey in a new tab, where it will look and work much like it did in the Survey.js design tool.

Week 6: Acceptability Judgement Data

This week we'll be dealing with some data from the reading, which includes large scale study on the alternative embedded passive. In short, the work addresses why some verbs can be used like *need* in (2) or (3) below (at least sometimes in some varieties), while others can only have embedded passives like in (1).

- The car needs to be washed
- The car needs washed
- The car needs washing

Dan Duncan has kindly shared data with us from the paper from UK respondents for some verbs. The whole dataset would be too large for us to deal with in Posit Cloud, but this subset of the data is still 9,000 rows. **You wouldn't want to deal with this data manually.**

I assume you have done the reading before attempting this exercise, since doing specific things in a specific order is sort of the point of everything we're doing here. Whether you have or not, you might want to pull up the paper in a tab next to the exercise to allow you to make some comparisons between the graphs you make here and what Duncan (2024) shows in the paper itself. Because much of the point of Duncan (2024) is that there isn't yet a complete description of AEP in the literature, a lot of his findings are more to do with exploring the set of acceptabliity judgements he collects than doing very targeted hypothesis testing. However, he does outline some specific ideas about how particular independent variables will affect his key dependent variable of (A)EP acceptability:

- **Semantic likeness category:** The category of a verb will determine how acceptable people find it in the AEP. In other words, some categories will have higher rates of acceptability than others, and the acceptability of verbs within a category will cluster together.
- **Syntactic type:** Raising verbs will have the highest acceptability ratings for AEP, followed by ECM; Control will have the lowest.
- **Productivity:** Verbs that generally have higher productivity in the embedded passive will have higher acceptability ratings.
- **Region:** AEP will generally be more acceptable for speakers of Scottish/Northern Irish English than for speakers from Wales/England (Duncan (2024) doesn't specifically focus on this, instead focusing on potential similarities/differences in US/UK Englishes).

Load the data

- Create a new script called `Week6_AEP.R` and save this into your scripts folder.
- Make sure you've loaded the tidyverse library at the top of the script.
- The data file should be in your data folder, where you moved it in the week 3 exercise. Load this into a variable for use in your analyses.
- Use `View()` and `summarize()` to understand what the data looks like, e.g., what are each of the columns and what do they mean?

Visualising trends

- Using the basic plotting templates below, add the missing variables so that each creates a reasonable plot.
  - You. need to define the x axis for the first two plots, and the y axis for the third plot.
  - For each plot, give it a and a transparent label for each axis using `xlab` and `ylab`, a reasonable title using `ggtitle` for and the plot overall.

- Once you've made just one of these, you'll see that the legend for the colour isn't quite ideal. Looking at the documentation for `scale_colour_manual()`, think about how you can fix this so that the legend reads England/Wales and Scotland/Northern Ireland (hint: use the `labels=c()` argument.)

This will give you one plot for each of the linguistic variables listed above (semantic likeness category, syntactic type, and productivity).

```
vproductivity<-ggplot(data=dfname, aes(x=, y=AJTResponse, colour=Region))+
  geom_point()+
  #make productivity log scale as in the original paperß
  scale_x_log10()+
  ggtitle()+
  xlab()+
  ylab()+
  theme_bw()

# save the plot to a file
ggsave("vproductivityPlot.png", width=10, height=8)

syntype<-ggplot(data=dfname, aes(x=,y=AJTResponse, colour=Region))+
  geom_violin()+
  ggtitle()+
  xlab()+
  ylab()+
  theme_bw()

# save the plot to a file
ggsave("syntypePlot.png", width=10, height=8)

slcat<-ggplot(data=dfname,aes(x=AJTResponse,y=,colour=Region))+
  geom_violin()+
  ggtitle()+
  xlab()+
  ylab()+
  theme_bw()

# save the plot to a file
ggsave("slcatPlot.png", width=10, height=8)
```

Part of your reflections will involve making some comparisons between these plots, so keep them handy!

Plotting means and standard deviations

The plots we've made above show some useful distributions, but they don't quite look like most of what is in the original paper. This is because this requires calculating means and standard errors for particular groupings of items; specifically, the mean and se of ratings for different syntactic types and semantic likeness groups (Figures 4 and 5 in Duncan, 2024).

- To calculate standard error, we'll use the `std.error()` function from the plotrix package. Install this package using the packages tab in the lower right pane, or type `install.packages("plotrix")` in the console.

- In your script, create a new tibble that will use `group_by` to group the data by `Region` and `SyntacticType`, and `summarise()` to give a table of the `mean()` of `AJTResponse` and the `std.error()` of `AJTResponse` for the relevant groups.
- Using the same strategy as above, create another tibble and do the same for semantic likeness categories (grouping by this variable instead of syntactic type).
- For each new tibble, check this has given you what you would expect by using the `View()` function.

Now, our aim will be to recreate Figures 4 and 5 from Duncan (2025), only contrasting regions within the UK rather than countries. Use the template below to make two separate plots in your script. Note:

- You'll need to change the name of the plot for each one, one for syntactic type and one for semantic likeness categories.
- The x aesthetic should be whatever you called your meaned AJTResponse when you used mutate to add a new column.
- The y aesthetic should be the relevant categorical variable for each plot (syntactic type and semantic likeness category)
- Make sure to add reasonable labels to the X and Y axes; you can borrow these from Duncan (2024) Figures 4-5. You may also want to use `scale_colour_manual()` to add reasonable legend labels for the regions.
- In `geom_errorbar()`, you need to define the maximum as the mean plus the standard error, and the minimum as the mean minus the standard error.
- Use `scale_colour_manual()` and `labels=c()` to give the legend labels more readable values.
- In the `ggsave()` call, which saves the plot to an image file, you need to give each plot a unique filename.

```
response_means<-ggplot(data=newtibble1, aes(x=, y=, colour=Region))+
  geom_point()+
  geom_errorbar(aes(xmax=,xmin=), width=0.25)+
  xlab()+
  ylab()+
  theme_bw()

# save the plot to a file
ggsave("namethisplot.png", width=10, height=8)
```

Now, you'll use these plots in completing your reflections.

Week 7: Lexical Decision Data

This week we'll look at data from the lexical decision task that was the focus of the reading. Next week we'll look at ALL data before embarking on a much needed break.

As we start to get more and more into looking at data from specific methods, **think about what kind of study you're more interested in designing for your final assignment**: are you more drawn to the kinds of questions we might want to address using LDT or another kind of decision task (questions about processing and properties of words or phrases), or, do you think you might have questions that are more suited to ALL (questions about learning, cognitive biases, and potentially cultural processes like iteration)?

Keep in mind that you can design a study that includes more than one method. Hudson-Kam and Newport (2005), a well-known ALL study which we'll go over in more detail next week, for example, uses ALL alongside acceptability judgments. We'll also look at ways we might combine ALL and decision tasks - I'll leave it to you to imagine how you might effectively combine AJT and LDT, but this shouldn't feel like a stretch.

This exercise will use data from the reading for this week, Sidhu, Pexman & Vigliocco (2019): Effects of Iconicity in Lexical Decision. We will deal with their data from Experiment 1 only.

Get the data

- SPV (2019) provide their data in an Open Science Framework repository, much like the Nettle data from week 2. Go to OSF and use search to find the relevant project
- Download the data from Experiment 1 (`Exp1.csv`) *and* their analysis file, `Effects of Iconicity in Lexical Decision Analyses.Rmd`. [hint]
- Remember that downloading an actual file requires a few clicks and isn't the same as the metadata for the file, even though the interface makes this easier to download.
- For most of this walkthrough, you'll mainly use the data file; but you will need look at and reference their analysis script in your reflections.
- Upload the data file to your project in the data folder, and their analysis file at the top level.
- Create a new R script in your scripts folder called `LexicalDecision_Week7.R`

Load and understand the data

- Make sure your script loads the tidyverse [hint], and read the data file into a tibble.
- Source the script to run it and then use the `View()` command in the console to eyeball the data and make sure it's loaded correctly solution.
- Use the `summary()` command in the console to take a closer look at the variables in the data. What are the variables and what are their types? Does anything jump out that isn't quite right? hint

While there are 240 trials per participant here, only some of them aren't really relevant; we don't really care about the non-word trials; these were used to force a decision. SVP also removed some other trials from the data for other reasons (hint)

- Use `filter()` to create a new tibble of only the trials that are relevant for analysis; don't worry about the constraint regarding removal of trials were a participant's RT was a certain distance from their own mean.

Make a plot

- Examine Figure 1 in the paper. It shows RTs on the y axis and Iconicity on the x axis, with a separate line for each block in the experiment.
- Use the skeletal ggplot code below (copying it into your script) to recreate a version of figure 1 from SVP. You can jazz it up and use different colours if you wish - I love this tool for finding colour palettes</> (tap the spacebar to generate a new palette). Note that even if you coloured the lines precisely as in the paper, it won't be exactly the same - you'll address why this is in your reflections.

```
#replace the tibblename, x and y varnames to align with what is in your tibble
lexplot<-ggplot(data=tibbleNameHere, aes(x=xvarhere, y=yvarhere))+
  geom_smooth(aes(colour=as.factor(Quarter), method='lm',se=FALSE))+
  xlab()+
  ylab()+
  scale_colour_manual(name="Trail Block",labels=c(),values=c())+
  theme_bw()

# show the plot in the lower right pane
show(lexplot)

# save the plot to a file
ggsave("myLexIconicityPlot.png", width=8, height=8)
```

Week 8: Artificial Language Learning Data

This week we'll look at data from the experiment we demoed in class, where we did a Stroop task and a very short artificial language learning (ALL) task. We'll only be looking at the ALL data.

Get the data

Unlike some of the published data we've dealt with in the module, this data is from a student project (like the colour data from weeks 3-4). However, note that the informed consent for both of these studies clearly stated that data would be used for teaching and research purposes.

- Download the `ALL_Responses.csv` file from this R Project and upload it into your own data folder.
- Create a new script in your scripts folder called `ALL_Week8.R`
  - Read the data file into a tibble. Source the script and then use the `View()` commannd in the console to eyeball the data and make sure it's loaded correctly.
- Use the `summary()` command in the console to take a closer look at the variables in the data.
- Returning to the full view of the data on the top left pane, scan down the `ProducedLabel` column in particular. What's wrong here? :solution

Clean the data

- We don't want responses where the participant entered "?". Use `filter()`to create a copy of the raw data tibble and remove remove these responses.

Note that this will work for our purposes in this exercise, but it's a bit too simple for a real approach to analysis. Ideally, we'd take the time here to make more principled decisions about what to remove and why. What the student actually did was to allow participants to provide "?" for up to two responses and then just remove those responses, but if a participant provided more than three question marks for either test block, this was taken as an indication that the participant wasn't paying attention to the task and we removed all of their data. For now, in the interest of moving on - and because we're not calculating any by participant measures - we'll use a quick filter only on the `ProducedLabel`. However, note that this was an illustrative moment for considering how to design a task that gets you valuable data: if I were to do this again, I would make it so that the participants could only enter alphabetic characters (and not punctuation) for labels.

Calculating edit distances

The next thing we want to do is calculate edit distances between the label the participant was trained on (`TargetLabel`) and the label they provided at test (`ProducedLabel`). We walked through how this is done in class, but we're not doing it manually: there's a package for that.

- In the console, type `install.packages(stringdist)`. Once that's done, you need to add `library(stringdist)` to the top of your script (below your standard `library(tidyverse)`). You only need to install it once, but you want to make sure the script loads it every time it runs.
- Use `?stringdist()` in the console to understand a bit about how the function works. What different methods are there? Which do yo think we want? :answer
- Using a pipe, add line that forces the data `rowwise()` so you can iterate over each row to find the difference between the target and produced labels.
- Now, pipe in another line which uses `mutate()` to add a column called `LevDist` that calculates the distance between the `TargetLabel` and the `ProducedLabel`

Normalising distances

Levenshtein distance is a count of the number of insertions, substutitions, or deletions needed to change one string into another, but these can't capture some relationships (e.g., as we discussed in lecture, *nid* and *rux* are 3 changes apart and so are *nidofrex* and *nitefrix*, but the latter pair is obviously closer than the former). To do this we need to normalize by the length of the longer string between the `TargetLabel` and `ProducedLabel`

- Use `str_length()` and `mutate()` to pipe in code which adds a column `MaxLen` with the length of the longer string by calling the `max()` function. You can use a new pipe/line and `mutate()` call, or you can add this to the existing mutate call. :hint
- Now you want to divide the `LevDist` in each row by `MaxLen.` Pipe in a new `mutate()` call that adds a column called `NormDist` defined as `LevDist/MaxLen`. Note that this has to be piped in as another `mutate()` statement and can't be merged with the earlier one :Why not?
- Check your new variable: because `NormDist` is a normalised variable, all values should be between 0 and 1. Use the `summarise` function in the console to verify this.

Summarise and plot

Now you want to make a summary of mean edit distances - your dependent variable - in terms of some independent variables (`TestBlock` and `Condition`), so that you can look at these graphically and see if anything interesting is going on. Specifically, the particiapnts did two blocks of testing: one after only seeing the form-meaning mappings once (`TestBlock` value `A`), and another after having seen the form-meaning mappings *twice*, as well as having undergone a round of testing which included feedback (`TestBlock` value `B`). What do you expect to happen to the distances between blocks and what does this mean?

- Pipe in an `ungroup()` statement. This is currently grouped `rowwise()`, but we want to calculate the mean gropuing it by `TestBlock` and `Condition`, so we have to ungroup it first.
- Pipe in a `group_by()` statement that groups the data by `TestBlock` and `Condition`, and a `summarise()` statement that defines a variable called `MeanDist` as the mean of the `NormDist` column across those groups. For this second part, use the handy `mean()` function.

How to submit your work

Once you've completed the week 1-8 exercises, double checking your reflections document and task checklist, it's time to submit your work. You won't submit this as a document like you would most assignments; you will download and submit your entire R Project as a zip file so that I can see your reflections *and* scripts you've written *and* plots you have made in your own organised little project.

To do this:

- Go to the files pane and select *all* the files and folders on the top level (e.g., `.Rhistory`, `project.Rproj`, your reflections document, the `Week5_SurveyDemo`, your `Scripts` and `Data` folders, and any other files or folders that are part of your project). Selecting a folder automatically selects all of the files inside it.
  - **Make sure you have everything selected** - part of what you will be marked on (in addition to your reflections) is evidence in this bundle of files that you have actually worked your way through the relevant exercises each week.
- Once everything is selected, go to the gear icon and click 'Export'. This will make a window pop up where you choose a name for a file ending in `.zip`. Make sure to keep the `.zip` file extension, and name your file `[STUDENTNUMBER]-SEL2229-Assignment1.zip` (replacing `STUDENTNUMBER` with your unique student number). Once it's appropriately named, download this file by clicking 'Download'.
- Go into the submission area for the assignment in Canvas, and submit the entire zip file by uploading it.