Introductory Notes to Using R

Trevor Chow

R is a programming language often used for statistics. This is a set of notes based on the ModernDive book - we will explore how to use R for statistical inference. We will begin with the basics of R and how the language works. Then we will look at how to use R in the data science pipeline - from importing and tidying data, to the processes of transformation/visualisation/modelling/statistical inference, before finally communicating it.

R Basics

There are different data types in R - integers, doubles/numerics, logicals, and characters.

- Integers are values like -1, 0, 2, 4092.
- Doubles or numerics are a larger set of values containing both the integers but also fractions and decimal values like -24.932 and 0.8.
- Logicals are either TRUE or FALSE.
- Characters are text such as "The Newsroom is the greatest TV show ever". They are denoted with the quotation marks around them.

Vectors are series of values created by the `c()` function, where `c(2,3,5,7)` would be a four element series. Factors are ways of representing categorical data. Data frames are rectangular spreadsheets.

We can test for equality using `==`, and do Boolean algebra using mathematical operators such as `>`, `>=` and `!=`. There are also the logical operators of `&` (and) as well as `|` (or).

We can explore data frames by using RStudio's data viewer with `View()`, using `glimpse()` in the `dplyr` package, using `kable()` in the `knitr` package (which is R Markdown friendly) and using `$` for looking at a single variable or column. Within a data frame, there are identification variables and measurement/characteristic variables.

Data Science with tidyverse

We can visualise data with `ggplot2`. A statistical graphic is a `mapping` of `data` variables onto the `aes`thetic attributes of `geo`metric objects. This can be be presented with `facet`ing the data into several plots, and with `position` adjustments for barplots.

For example, we can take a hypothetical `data_frame` and plot it via `ggplot(data = data_frame, mapping = aes(x = independent_var, y = dependent_var)) + geom_point()` to get a scatterplot. Because the points may overlap and cause overplotting, we can specify `geom_point(alpha = 0.2)` to increase the transparency of each point or jitter the points by `geom_jitter(width = 10, height = 10)`. We can get a linegraph with `geom_line()`.

We can get a histogram with white borders and blue steel bins using `geom_histogram(color = "white", fill = "steelblue")`. The bins can be adjusted by using the `bins` argument or the `binwidth` argument.

We can add `+ facet_wrap(~ month, nrow = 4)` to separate the plot into one plot for each month across 4 rows, if `month` is in the data frame.

A boxplot can be created via `geom_boxplot()` - but for categorical variables, we can do `ggplot(data = data_frame, mapping = aes(x = factor(categorical_var), y = dependent_var)) + geom_boxplot()`.

Finally, we have barplots - if the categorical data is listed individually, we use `geom_bar()`, while if it is pre-counted, we use `geom_col()`.

This gets more complicated with multiple categorical variables - we can use `ggplot(data = data_frame, mapping = aes(x = first_var, fill = second_var)) + geom_bar(position = "dodge")` to get side-by-side barplots or use `facet_wrap(~ second_var)` to get a faceted barplot.

We can wrangle data using `dplyr`. The pipe operator `%>%` takes the output of the previous function and inputs it into the next function. The `filter()` function filters out the data based on some criteria regarding the values of variables.

We can find various summary statistics by `summary_values <- data_frame %>% summarise(mean = mean(some_var, na.rm = TRUE), std_dev = sd(some_var, na.rm = TRUE))`, which removes all missing values. There are other summary statistics, such as `max()`, `min()`, `IQR()`, `sum()` and `n()`.

We can group observations by the value of another variable using the function `group_by(first_var, second_var)`. We can transform data by using `mutate()`, such as `data_frame <- data_frame %>% mutate(new_var = second_var - first_var)`. To order data, we can pipe a data frame into `arrange()` or `arrange(desc())`, which will sort it by some variable.

Data frames can be merged, with `data_frame_joined <- first_data_frame %>% inner_join(second_data_frame, by = "some_var")`. If we want to join data frames with variables that have different names in the two data frames, we can use `by = c("first_name" = "second_name")`. By contrast, if we want to join data frames by multiple key variables (because that is required to uniquely identify each observational unit), we can use `by = c("first_var","second_var","third_var")`.

Other ways of wrangling data including using `select()` to pick a subset of variables or columns - the argument `-first_var` excludes `first_var`, `first_col:nth_col` selects the range of columns from `first_col` to `nth_col`, and the helper functions of `starts_with()`, `ends_with()` and `contains()` can be used. The `rename(new_name = old_name)` function can rename variables, while we can select the first `n` values of a variable using `top_n(n = 10, wt = chosen_var)`.

We can import data with `readr`. We can import a csv using `read_csv()`. We then need to tidy the data - tidy data is where each variable forms a column, where each observation forms a row, and where each type of observational unit forms a table. We can convert to tidy format using with the `tidyr` package and the function `pivot_longer(names_to = "", values_to = "", cols = )`.

For more, check out the cheatsheets on the Rstudio website.

Data Modeling with moderndive

We can model for explanation or for prediction - that is, make explicit the relationship between an outcome/dependent variable y and an explanatory/predictor/independent variable x. The `tidyverse` package includes a set of data wranggling packages, the `moderndive` package allows linear regression and `skimr` can compute summary statistics.

Before doing any modeling, we should perform an exploratory data analysis. This involves looking at raw data values with `glimpse()`, computing summary statistics with `skim()` and `get_correlation()` as well as creating data visualisations.

To begin with, we have a numerical outcome variable and a single numerical explanatory variable - the following plots them out alongside a regression line based on a linear model, with standard error uncertainty bars.

```
ggplot(data_frame, aes(x = explanatory_var, y = outcome_var)) +
  geom_point() +
  labs(x = "Explanatory", y = "Outcome",
       title = "Relationship between explanatory and outcome") +
  geom_smooth(method = "lm", se = TRUE)
```

We can then analyse the data.

```
model <- lm(outcome_var ~ explanatory_var, data = data_frame)
get_regression_table(model)
```

We can repeat this process with a numerical outcome variable and a single categorical explanatory variable. We can make a regression table and use that to produce an equation with indicator functions. We can examine the specifics of the model by looking at the points.

```
regression_points <- get_regression_points(model)
```

For a numerical outcome variable with two explanatory variables, one numerical and one categorical, we can plot it and use colours to distinguish, and then fit the model to an interaction model.

```
ggplot(data_frame, aes(x = num_var, y = outcome_var, color = cat_var)) +
  geom_point() +
  labs(x = "Numerical", y = "Outcome", color = "Categorical") +
  geom_smooth(method = "lm", se = FALSE)

model <- lm(outcome_var ~ num_var * cat_var, data = data_frame)
get_regression_table(model)
```

In some cases, we may want to use a parallel slopes model.

```
ggplot(data_frame, aes(x = num_var, y = outcome_var, color = cat_var)) +
  geom_point() +
  labs(x = "Numerical", y = "Outcome", color = "Categorical") +
  geom_parallel_slopes(se = FALSE)

model <- lm(outcome_var ~ num_var + cat_var, data = data_frame)
get_regression_table(model)
```

These processes are also possible when it is two numerical explanatory variables instead.

Statistical Inference with infer

We take samples of a population using `rep_sample_n(size = 25, reps = 100)`. We can then plot the sampling distributions and calculate point estimates. The standard deviation of point estimates is called the standard error.

As the sample size increases, the standard error will decrease and lead to a more precise estimate, as well as the sampling distribution more closely following the normal distribution.

We can engage in bootstrap resampling with replacement - that is, we resample the single sample from the population multiple times.

```
sample_data %>%
rep_sample_n(size = 50, replace = TRUE, reps = 1000) %>%
group_by(replicate) %>%
summarise(mean_value = mean(value))
```

We can construct confidence intervals with the percentile method, where a certain percentage of the resampled values fall in that range. We can also express it in terms of the standard error - however, this is dependent upon the bootstrap distribution being roughly normal.

Let's apply the tools of `infer`, which are more powerful than the `dplyr` tools we have used so far. We `specify()` the variables of interest, `generate()` replicates and `calculate()` summary statistics.

```
sample_data <- data %>% rep_sample_n(size = 50)

sample_data %>%
specify(formula = value ~ NULL) %>%
generate(reps = 1000) %>%
calculate(stat = "mean")
```

We can pipe the distribution into `get_confidence_interval(level = 0.95, type = "percentile")` and print it, or add the argument `+ shade_ci(endpoints = )`. The alternative is possible with `get_confidence_interval(type = "se", point_estimate = )`

Another method of statistical inference is hypothesis testing. This compares a null hypothesis H_0 with an alternative hypothesis H_1, with the null hypothesis normally being the claim of no effect. We look at the null distribution of the test statistic and compare it to the observed test statistic - the p-value quantifies the probability of obtaining a test statistic just as or more extreme than the observed value assuming the null hypothesis to be true. We compare it to the significance level α - if it is smaller than α, we reject the null hypothesis.

We can use another `infer` workflow to do so - we have a choice whether we are testing a point hypothesis or an independence hypothesis.

```
null_distribution <- data_frame %>%
specify(formula = outcome_var ~ explanatory_var, success = "") %>%
hypothesize(null = "independence") %>%
generate(reps = 1000, type = "permute") %>%
calculate(stat = "diff in props", order = c("male", "female"))

obs_diff_prop <- data_frame %>%
specify(formula = outcome_var ~ explanatory_var, success = "") %>%
calculate(stat = "diff in props", order = c("male", "female"))

visualize(null_distribution, bins = 10) + shade_p_value(obs_stat = obs_diff_prop, direction = "right")

null_distribution %>% get_p_value(obs_stat = obs_diff_prop, direction = "right")
```

There are two types of errors - Type 1 is rejecting H_0 when it is true, while Type 2 is failing to reject H_0 when it is false. The probability of Type 1 is the significance level of the test α, while the probability of Type 2 is β, while the power of the test is $1 - \beta$. In practice, we fix α and attempt to minimise β.

We can conduct inference for regression based on 4 conditions of LINE. Firstly, the linearity of relationship between variables. Secondly, the independence of the residuals. Thirdly, the normality of the residuals. Fourthly, equality of variance of the residuals. If instead the spread of the residuals increases as the value of the explanatory variable does, it displays heteroskedasticity and does not fulfil the criterion.