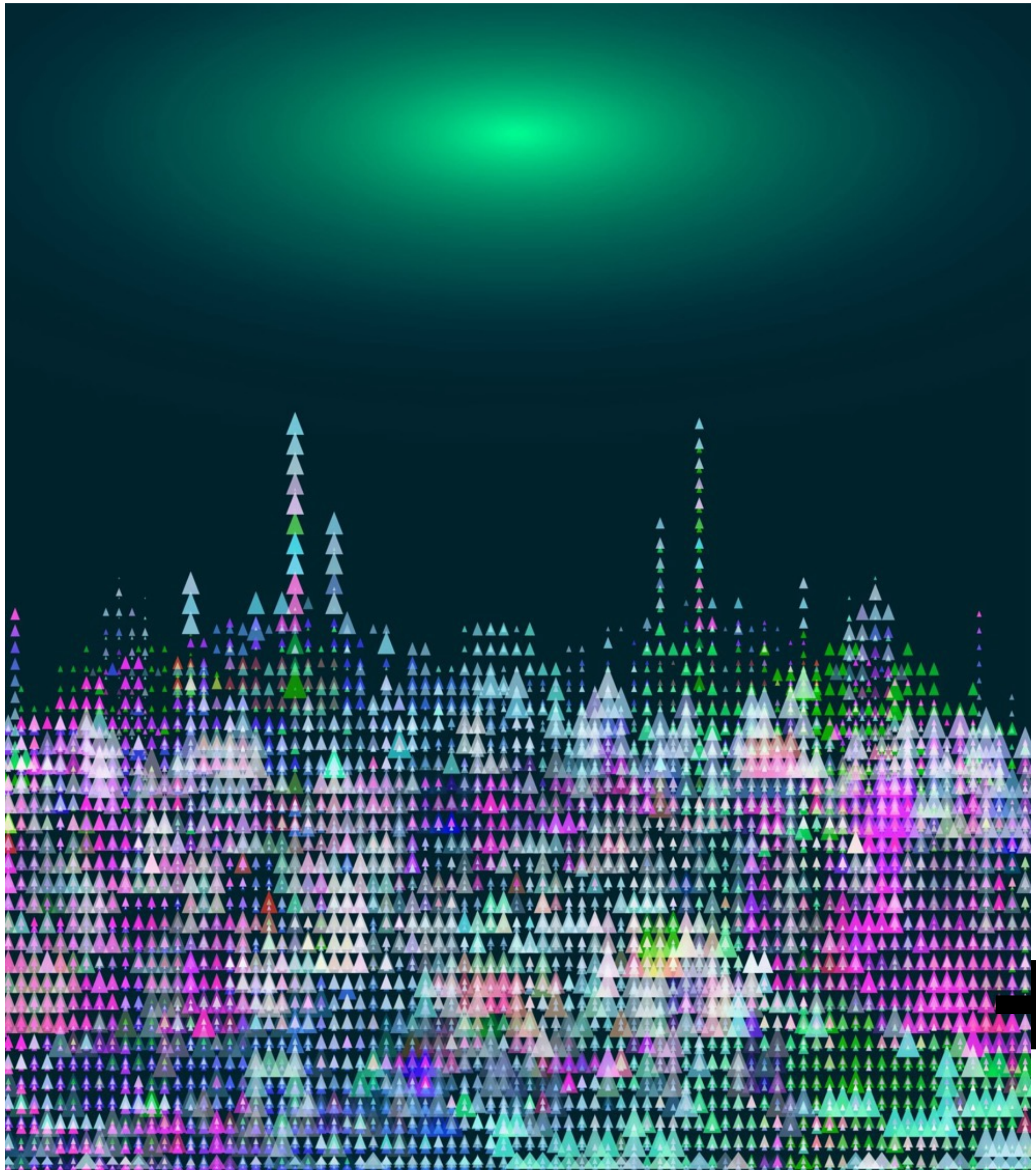


# Logistic & Poisson Regression

Quantitative Analysis

Week 10

## Small Mistakes

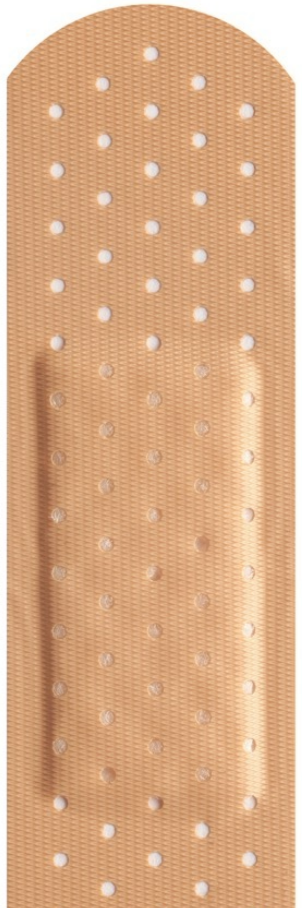
- **Your code should run!** Many of you didn’t include the `read_csv()` command to import the data. In some situations (such as when your code is being graded by an automated system, which many courses now use), this could result in you failing the assignment.

- Many commands in R will show you the result of an operation but won’t change any data in your data frames without explicitly being told to do so. For example, just typing `factor(my_df$my_col)` will show me the factors in that column – but to actually change that column into a categorical variable, you need a command that puts those factors back into the data frame:

```
my_df <- my_df %>%
        mutate(my_col = factor(my_col))
```

## Mistakes can Snowball

- Make one small error, and its effects cascade through the analysis, ruining the rest of your efforts.
- Some of you specified a model slightly wrong (a common problem was using an interaction effect without including individual variables), and that led you to make the wrong graphs and interpret the wrong results.
- Others mixed up variables – for example, creating a new column with an updated version of some data, but then using the old column name in model specifications. In some cases that stopped code from working at all.



## What is your Outcome Variable?

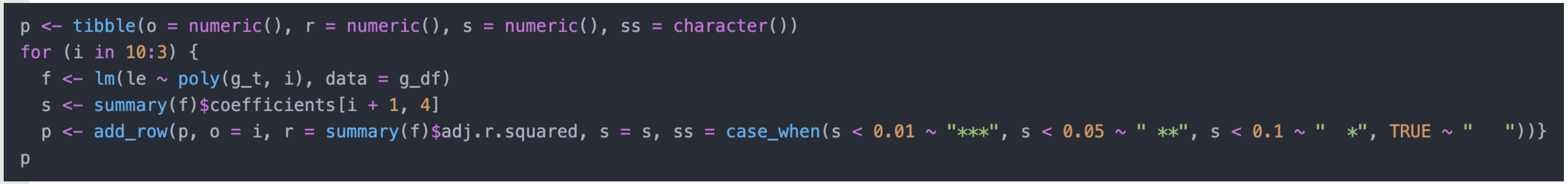
- Be clear at all times when creating models or writing R commands which variable is your **outcome variable** (sometimes called **dependent variable** or **response variable**).
- Especially when making visualisations, it's important to make sure you assign the correct variable to the Y-value or the "response" parameter – otherwise your graph will be hard to interpret at best, and outright wrong at worst.

## When to use Categories / Factors

- We can convert continuous variables into factors by using commands like `ifelse()` or `case_when()` inside a `mutate()` statement – for example, turning a salary variable into “High” or “Low”, or turning age into ranges (18-25, 26-35, etc.).
- Only do this when it’s necessary! Sometimes it can really help with making a model more interpretable – e.g., being able to see that a certain age group has a certain coefficient can be more intuitive than seeing that a 1-year increase in age leads to a 0.003 increase in the outcome.
- However, be aware that this conversion is losing some detail from your data. If there’s no good reason to convert a continuous variable into a category, leave it alone.

## Code Readability

- **You spend more time reading code than writing code.**
- There are a few simple ways to make your code readable, both for yourself and for others:
  - Give it **room to breathe** – split lines and put space between commands and arguments.
  - **Name your variables and functions** in a way that makes their purpose very clear.
    Don’t abbreviate. Never use single letter variable names.
  - **Add comments** for anything not immediately apparent.
    You’ll be grateful when you want to re-use the code for another project later – it sucks to have to Google a bunch of commands just to understand code you wrote yourself only a few months ago!



```r
p <- tibble(o = numeric(), r = numeric(), s = numeric(), ss = character())
for (i in 10:3) {
  f <- lm(le ~ poly(g_t, i), data = g_df)
  s <- summary(f)$coefficients[i + 1, 4]
  p <- add_row(p, o = i, r = summary(f)$adj.r.squared, s = s, ss = case_when(s < 0.01 ~ "***", s < 0.05 ~ " **", s < 0.1 ~ "  *", TRUE ~ "   "))}
p
```

## Example: Bad Code

This code works perfectly and does what it's supposed to do. It's also terrible.

Can you tell at a glance what the purpose of this code is?

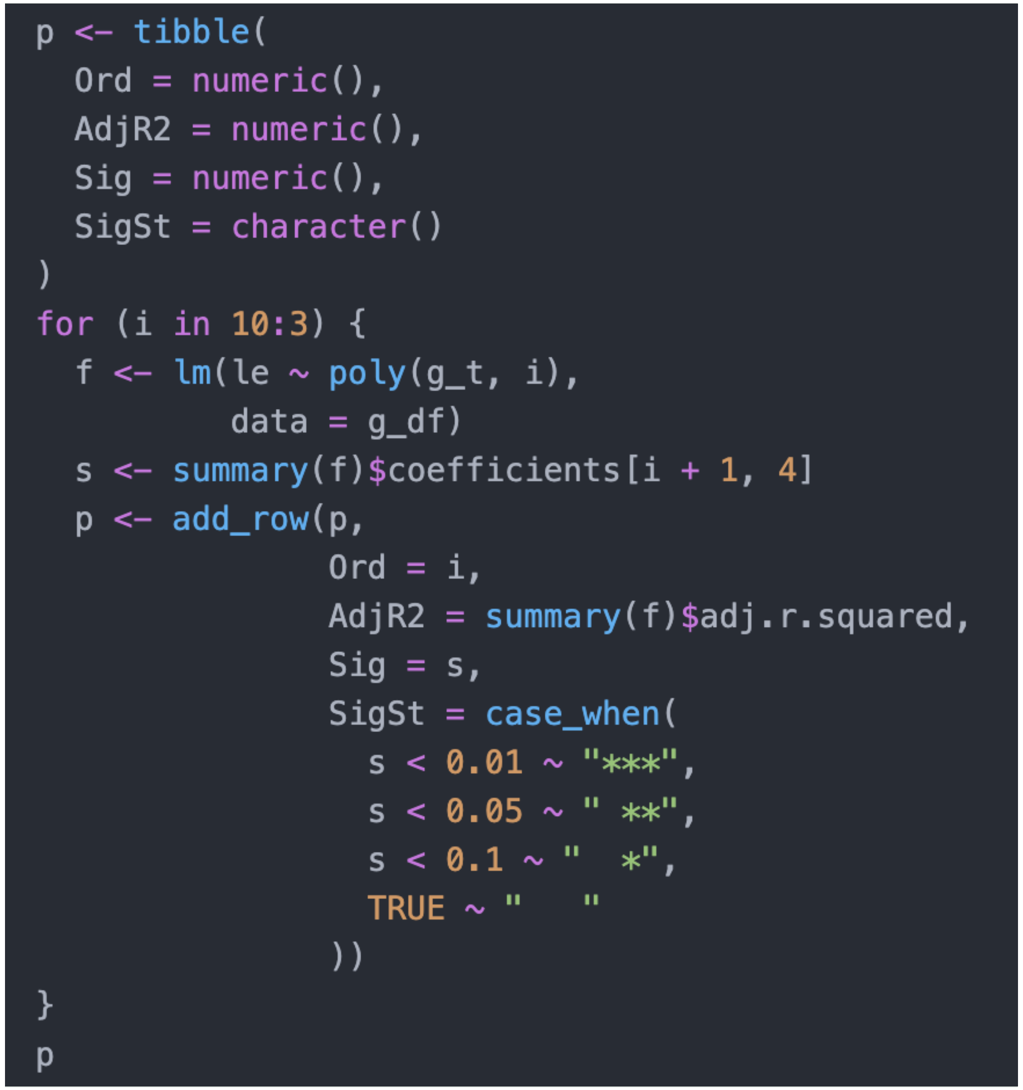
It's obviously something to do with regression – you can see stars being set for significance levels, and mention of coefficients. But beyond that, who knows what this code was written for?

# Example 2: Better (but still bad)

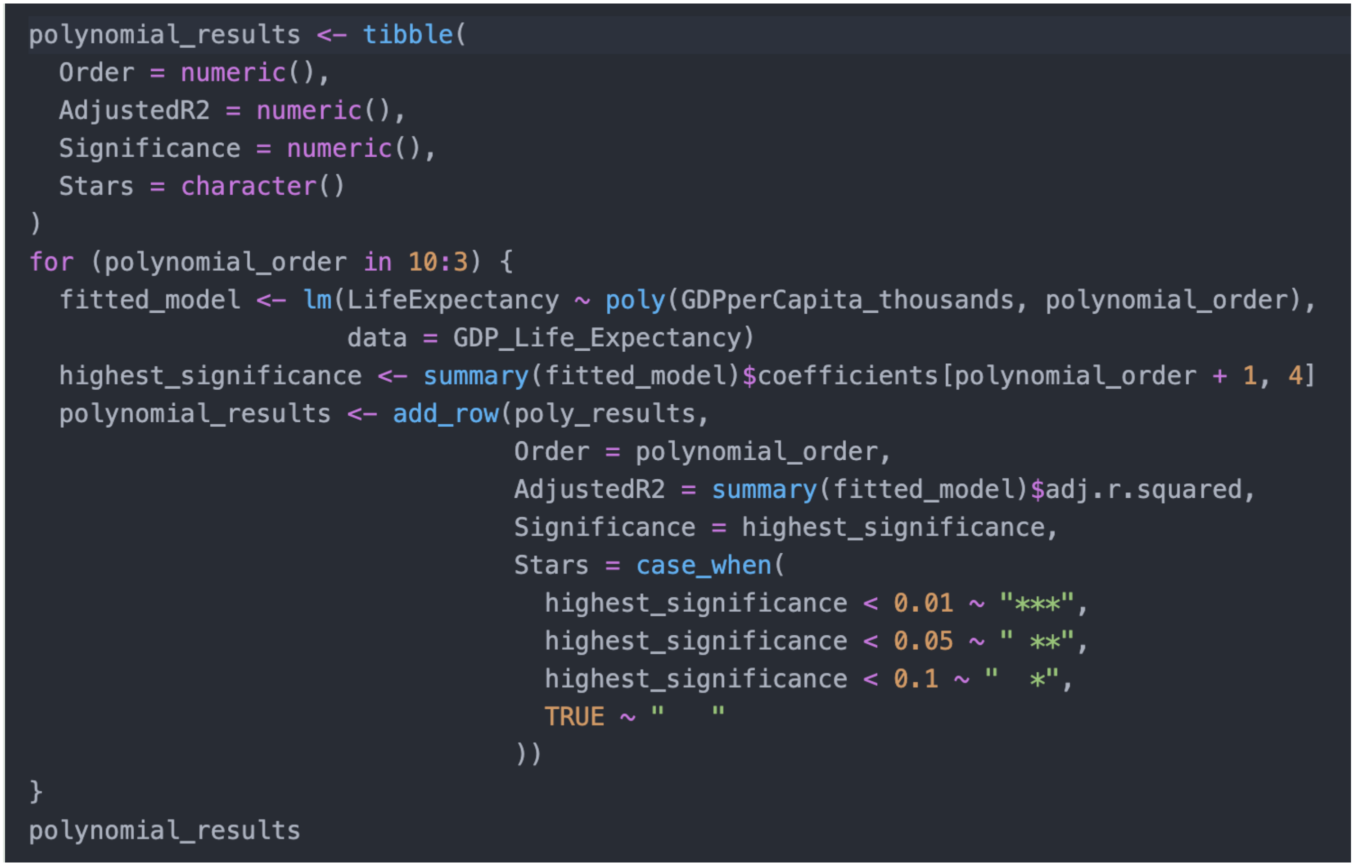
This is exactly the same code, with just two minor changes:

- the commands have been split out across lines;
- the table has slightly more sensible (but heavily abbreviated) column names.

It's still hard to tell what this code was meant to do, but at least it's easier – if still time-consuming – to go through it step by step and understand the process.



```r
p <- tibble(
  Ord = numeric(),
  AdjR2 = numeric(),
  Sig = numeric(),
  SigSt = character()
)
for (i in 10:3) {
  f <- lm(le ~ poly(g_t, i),
          data = g_df)
  s <- summary(f)$coefficients[i + 1, 4]
  p <- add_row(p,
               Ord = i,
               AdjR2 = summary(f)$adj.r.squared,
               Sig = s,
               SigSt = case_when(
                 s < 0.01 ~ "***",
                 s < 0.05 ~ " **",
                 s < 0.1 ~ "  *",
                 TRUE ~ "   "
               ))
}
p
```



```r
polynomial_results <- tibble(
  Order = numeric(),
  AdjustedR2 = numeric(),
  Significance = numeric(),
  Stars = character()
)
for (polynomial_order in 10:3) {
  fitted_model <- lm(LifeExpectancy ~ poly(GDPperCapita_thousands, polynomial_order),
                    data = GDP_Life_Expectancy)
  highest_significance <- summary(fitted_model)$coefficients[polynomial_order + 1, 4]
  polynomial_results <- add_row(poly_results,
                              Order = polynomial_order,
                              AdjustedR2 = summary(fitted_model)$adj.r.squared,
                              Significance = highest_significance,
                              Stars = case_when(
                                highest_significance < 0.01 ~ "***",
                                highest_significance < 0.05 ~ " **",
                                highest_significance < 0.1 ~ "  *",
                                TRUE ~ "   "
                              ))
}
polynomial_results
```

## Example 3: Good Code

The only thing that has changed here is the variable names.

There are no longer any single-letter variables, or any unnecessarily abbreviated variables.

Every variable and column has a name that precisely describes what it contains. Where relevant, the name also contains the **unit** – e.g., `GDPperCapita_thousands`.

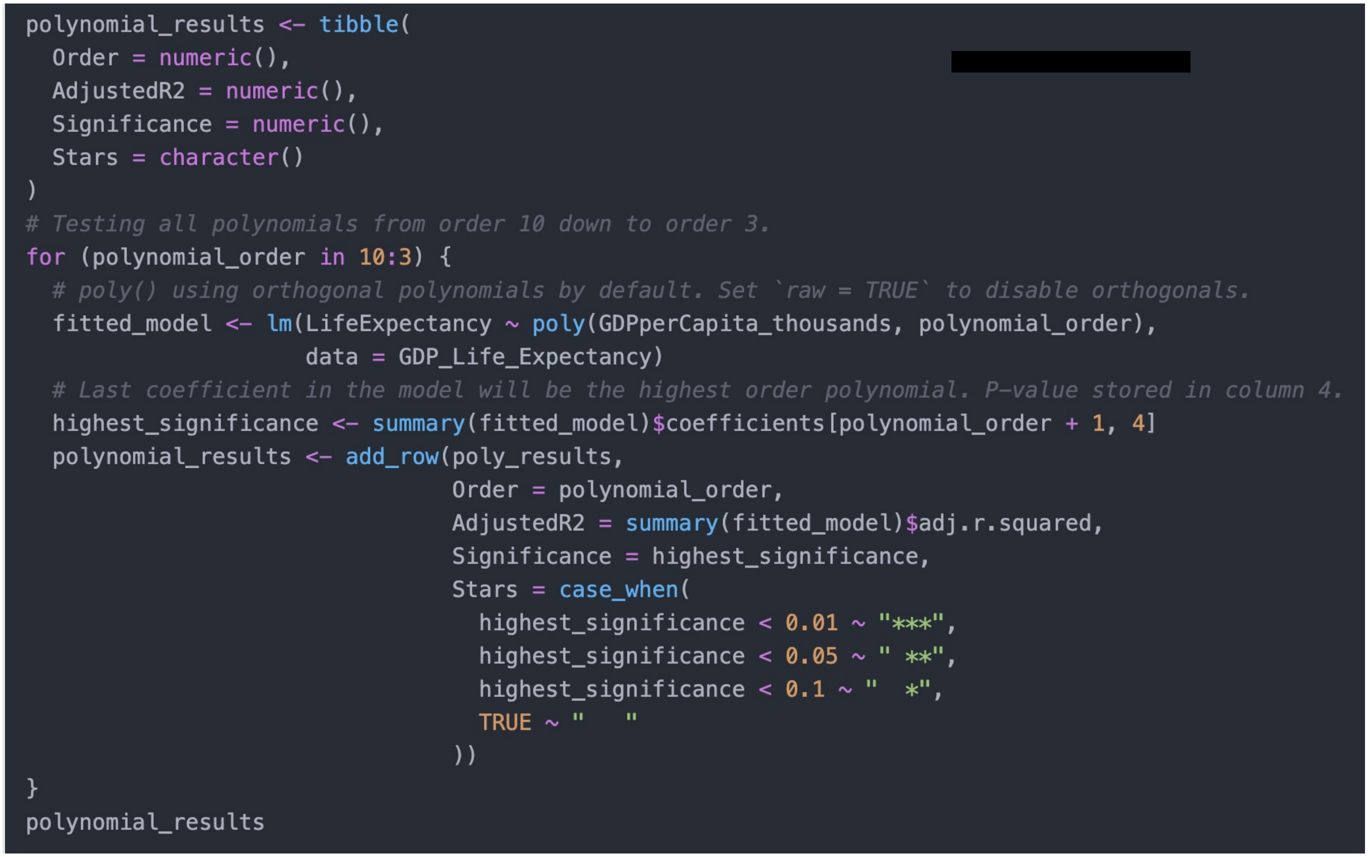
You can tell at a glance what this code does and could easily reuse or repurpose it in future projects.

## Example 4: Best Code

This code adds three short comments which make the purpose of the code easier to understand and could save you a ton of time in future when you need to reuse the code.

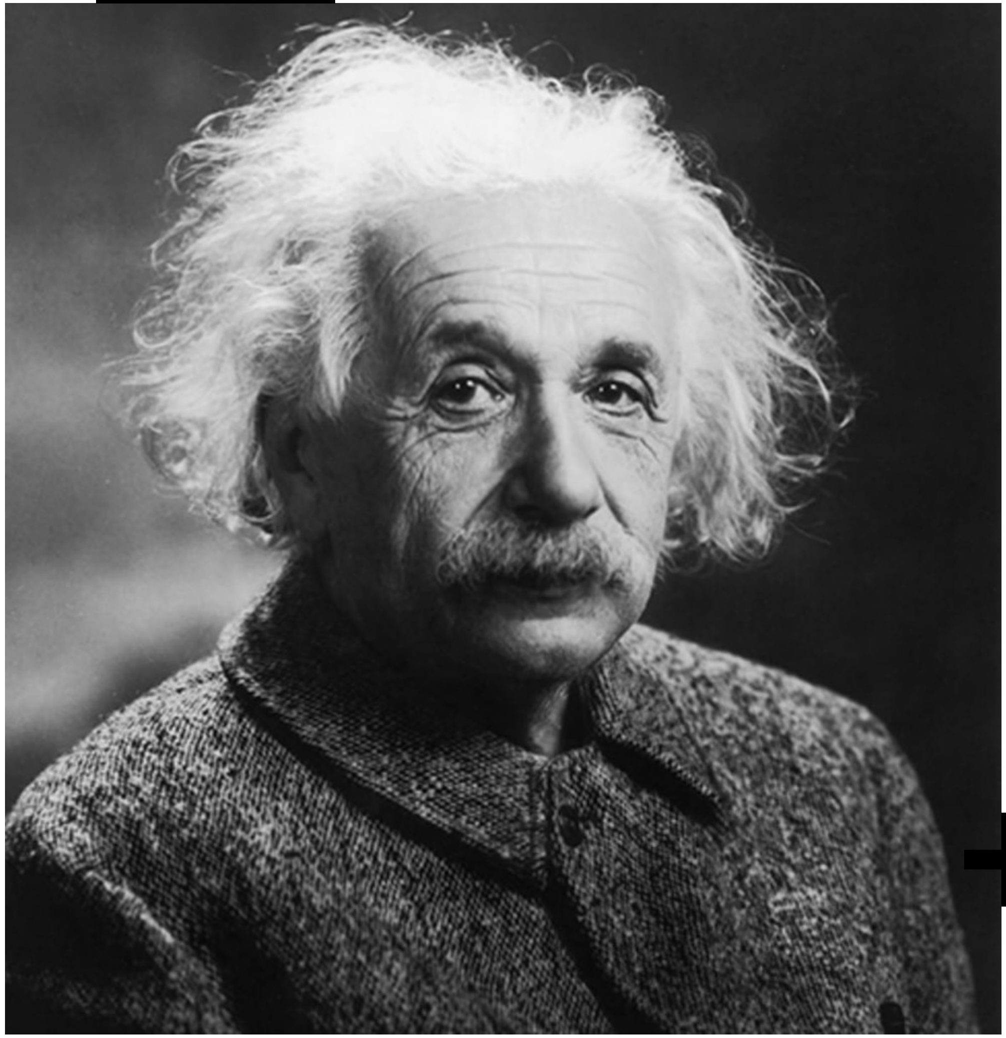
By documenting any little tricky aspects – such as poly() defaulting to orthogonal polynomials, as discussed last week – you save yourself having to Google commands you wrote to understand why you wrote them that way.

In a month you won't remember that poly() defaults to orthogonals – so leave yourself a hint.



```r
polynomial_results <- tibble(
  Order = numeric(),                                                          [redacted]
  AdjustedR2 = numeric(),
  Significance = numeric(),
  Stars = character()
)
# Testing all polynomials from order 10 down to order 3.
for (polynomial_order in 10:3) {
  # poly() using orthogonal polynomials by default. Set `raw = TRUE` to disable orthogonals.
  fitted_model <- lm(LifeExpectancy ~ poly(GDPperCapita_thousands, polynomial_order),
                     data = GDP_Life_Expectancy)
  # Last coefficient in the model will be the highest order polynomial. P-value stored in column 4.
  highest_significance <- summary(fitted_model)$coefficients[polynomial_order + 1, 4]
  polynomial_results <- add_row(poly_results,
                                Order = polynomial_order,
                                AdjustedR2 = summary(fitted_model)$adj.r.squared,
                                Significance = highest_significance,
                                Stars = case_when(
                                  highest_significance < 0.01 ~ "***",
                                  highest_significance < 0.05 ~ " **",
                                  highest_significance < 0.1 ~ "  *",
                                  TRUE ~ "   "
                                ))
}
polynomial_results
```

## Finding the Best Model



–“Make everything as
simple as possible,
but not simpler.”

## The Principle of Parsimony

- Also known as **Occam’s Razor**, or much more informally, as the **KISS Principle** (“Keep It Simple, Stupid”).
- All other things being equal, the simplest explanation, requiring the least elements, is the best explanation.
- In other words: do not add complexity to a model unless doing so produces a significant improvement. If you can achieve the same results with a simpler model, always use the simpler model.

## Basic Model Comparison

| | *Dependent variable:* | | | |
|---|---|---|---|---|
| | alison | | | |
| | None | Self:Age | Self:Country | Both |
| | (1) | (2) | (3) | (4) |
| self | $0.178^{***}$ | $0.165^{*}$ | $0.093^{**}$ | -0.012 |
| | (0.031) | (0.089) | (0.047) | (0.112) |
| countryMexico | $-1.117^{***}$ | $-1.116^{***}$ | $-1.450^{***}$ | $-1.493^{***}$ |
| | (0.081) | (0.082) | (0.161) | (0.166) |
| age | -0.003 | -0.004 | -0.003 | -0.007 |
| | (0.002) | (0.005) | (0.002) | (0.005) |
| self:age | | 0.0003 | | 0.002 |
| | | (0.002) | | (0.002) |
| self:countryMexico | | | $0.148^{**}$ | $0.172^{***}$ |
| | | | (0.062) | (0.066) |
| Constant | $3.668^{***}$ | $3.694^{***}$ | $3.862^{***}$ | $4.075^{***}$ |
| | (0.150) | (0.225) | (0.170) | (0.267) |
| Observations | 781 | 781 | 781 | 781 |
| $R^2$ | 0.281 | 0.281 | 0.287 | 0.288 |
| Adjusted $R^2$ | 0.279 | 0.278 | 0.283 | 0.283 |
| Residual Std. Error | 1.007 (df = 777) | 1.007 (df = 776) | 1.003 (df = 776) | 1.003 (df = 775) |
| F Statistic | $101.412^{***}$ (df = 3; 777) | $75.969^{***}$ (df = 4; 776) | $77.953^{***}$ (df = 4; 776) | $62.579^{***}$ (df = 5; 775) |
| *Note:* | | | | $^{*}$p<0.1; $^{**}$p<0.05; $^{***}$p<0.01 |

R’s output includes key **goodness of fit** measures, which tell us which model is best.

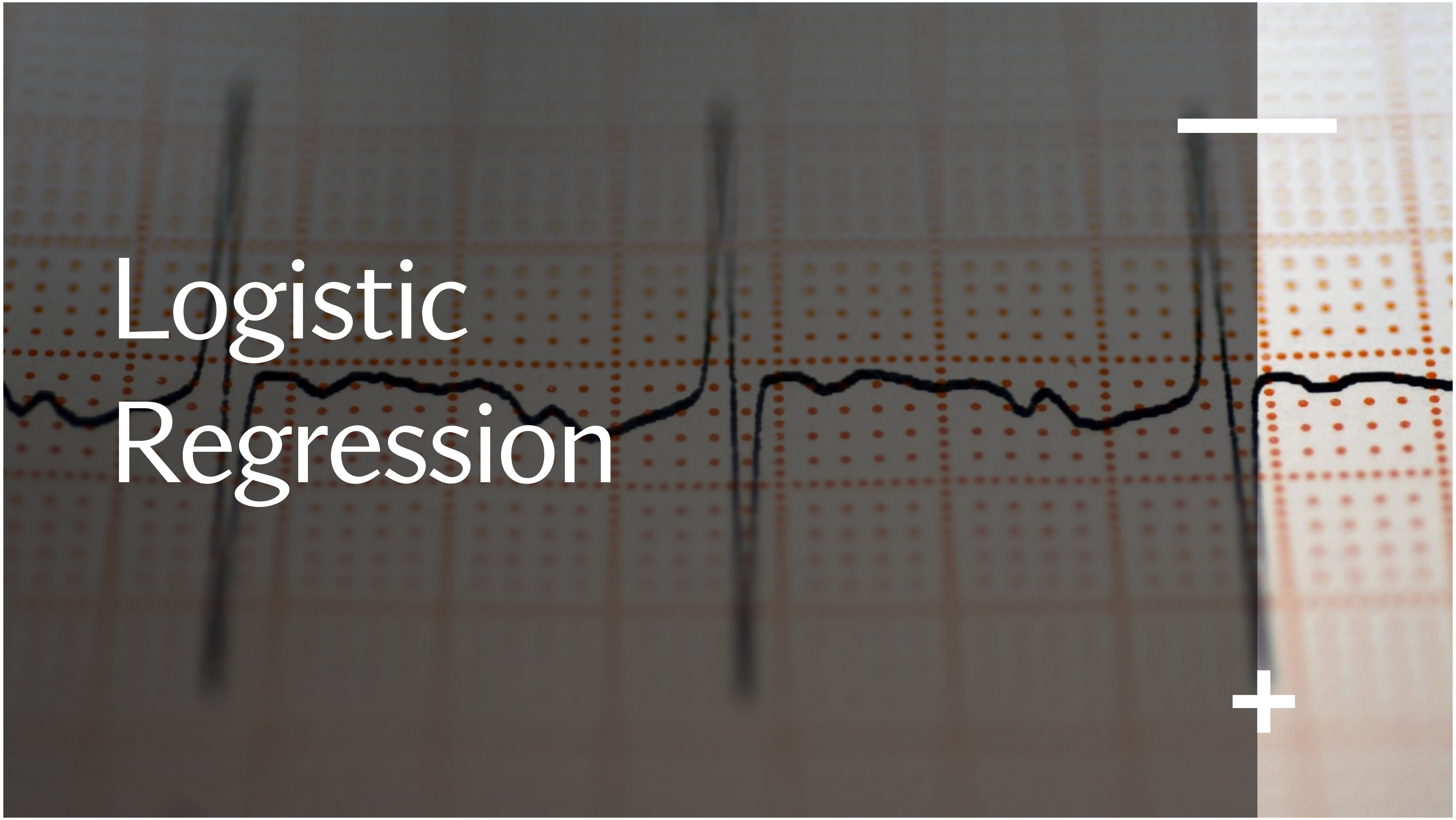
**Adjusted R²** is better when higher; **Residual Standard Error** is better when lower.

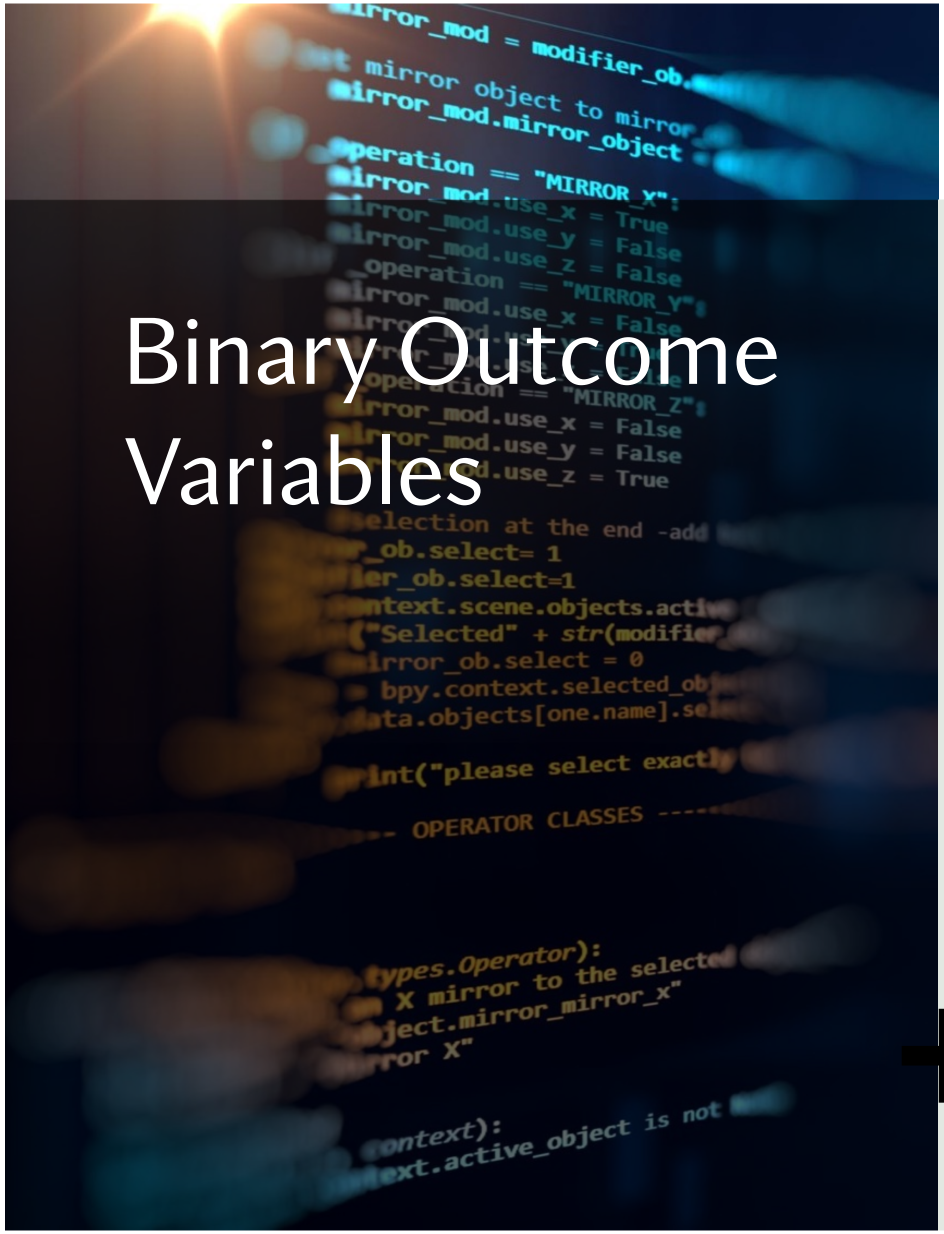
You can’t directly compare the **F Statistic**, but if it isn’t statistically significant, the model probably isn’t valid.

## Comparing Models with ANOVA

- Another approach that allows us to mathematically compare the quality of two models is to use a variant of the ANOVA test we saw a few weeks ago.
- By using ANOVA to compare models, we can see whether there is a statistically significant improvement in the fit of a more complex model compared to a less complex model; i.e., is it worth adding this complexity?

## Logistic Regression





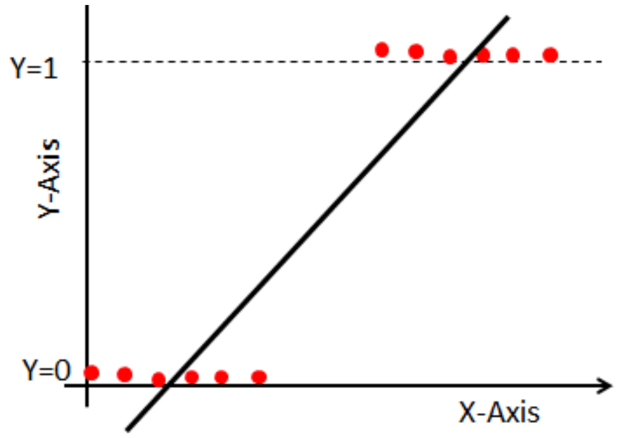
## Binary Outcome Variables

- Let's finally fill in one major gap in our knowledge of regression – how to handle non-continuous outcome variables.
  - In other words, how can we make a regression model when the outcome variable is something like "Yes" vs. "No"?
- Obviously, a binary variable can't really be normally distributed (it can only have two values, and there are no "outliers"), so the assumptions we use for OLS linear models do not hold.

**The problem is that since the outcome (Y) can only have two values (0 or 1), a linear regression will look like the one below.**

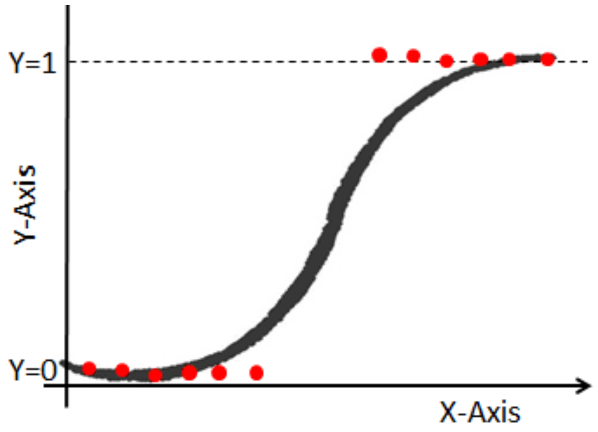
**It will try to predict a lot of values that aren't 0 or 1, because it doesn't understand that those are the hard limits of this variable.**

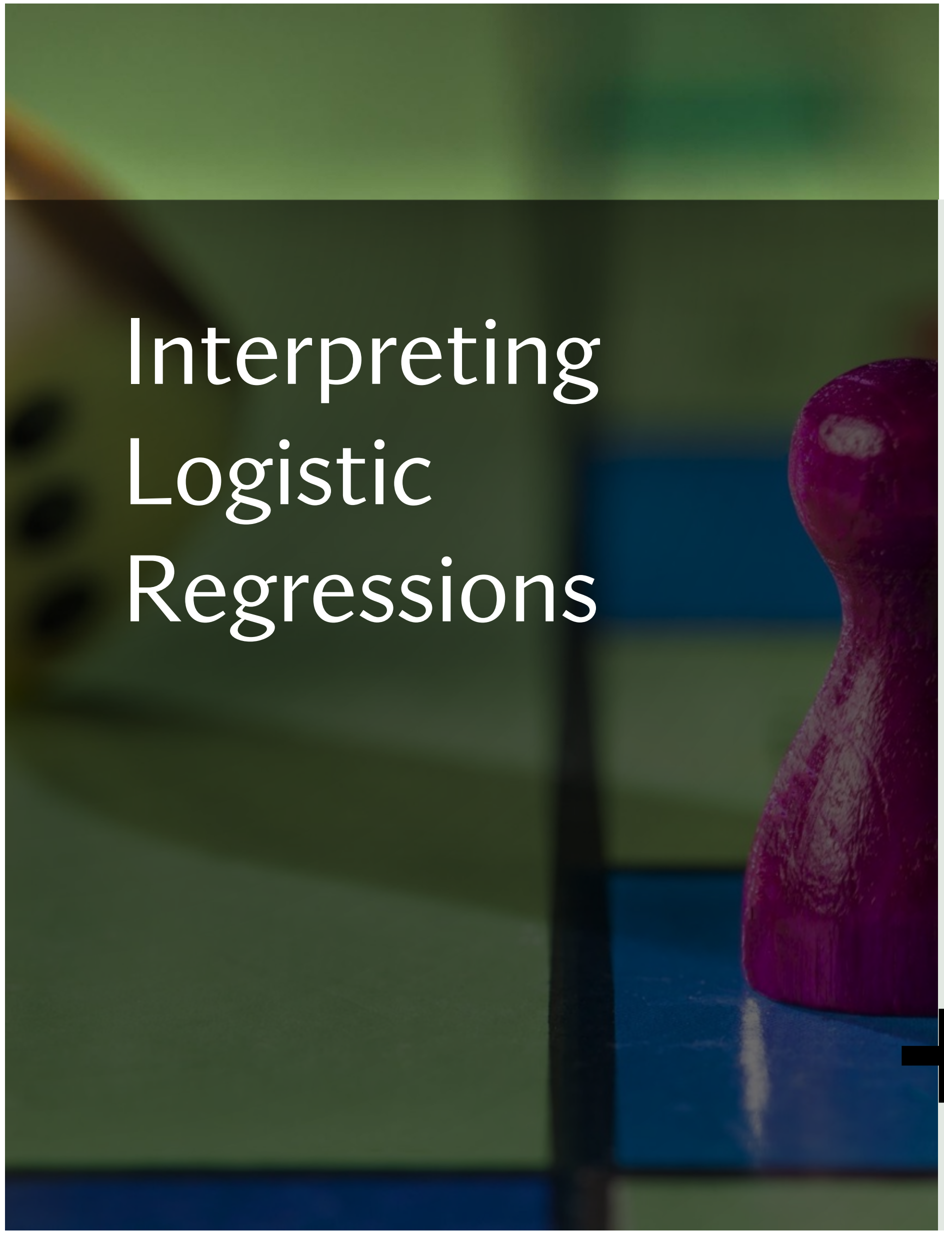
**It will also assume that most predicted values should be between 0 and 1, which of course isn't really possible.**



**The solution is to run a logistic regression. Instead of a straight line, this type of regression models a kind of S-curve, with limits at 0 and 1, and a sharp turn upwards in the middle.**

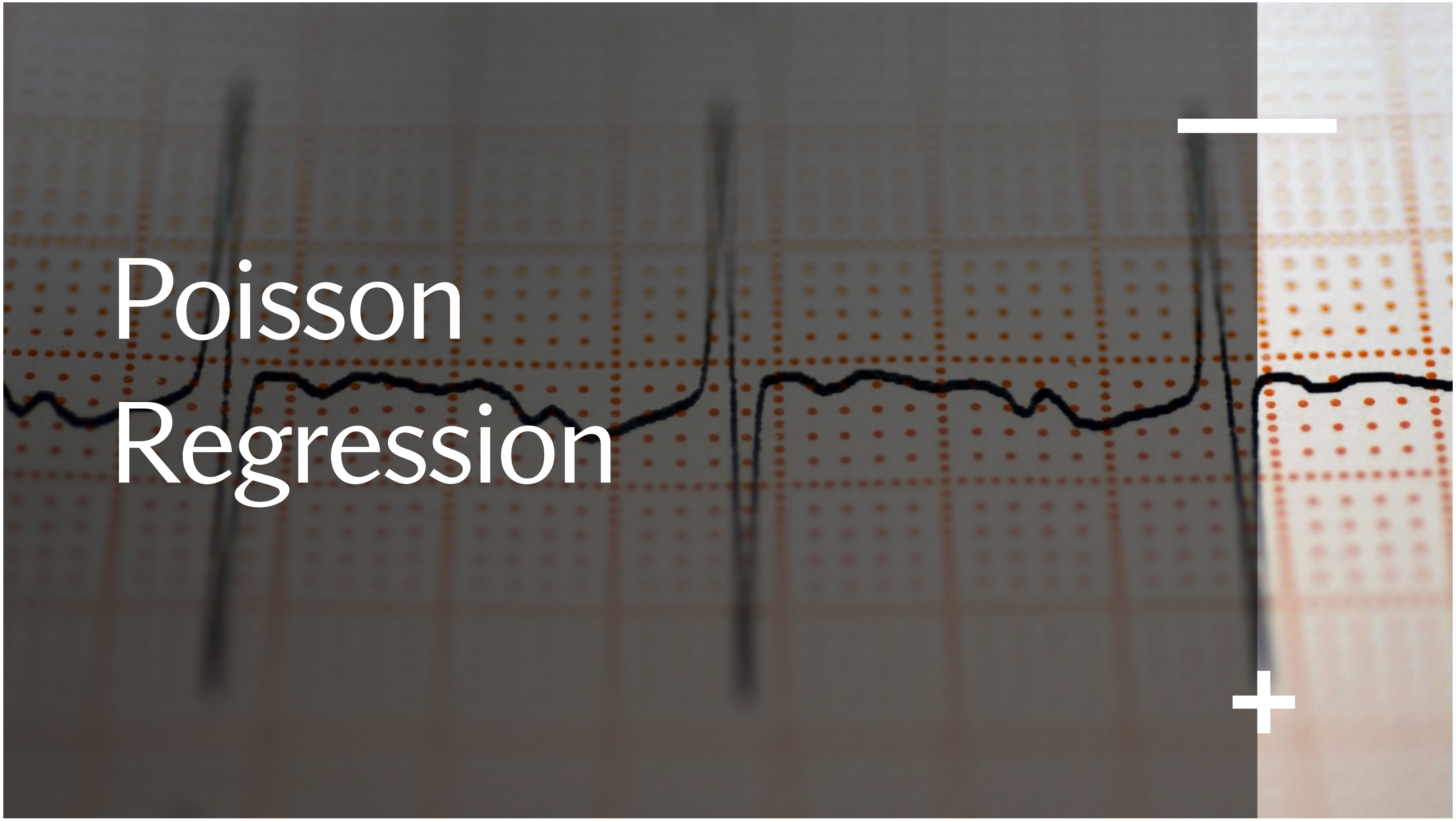
**Instead of predicting a continuous number, this model is calculating the probability of finding a 1 or a 0.**





## Interpreting Logistic Regressions

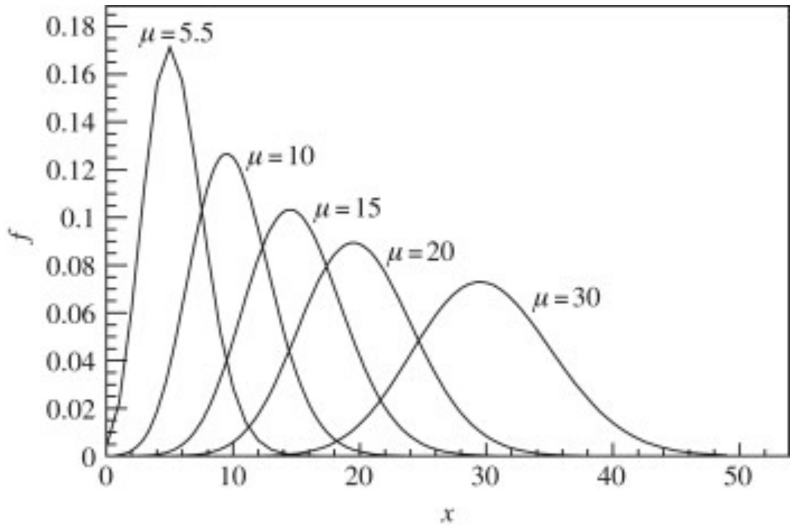
- Logistic Regressions produce probabilities of Y being 0 or 1, not predicted values of Y itself.
- These probabilities are expressed in a form called **log odds**. We can easily convert these to **odds**.
- **Odds** might be familiar to you from things like sports betting: 7/3, or 3/1, for example.
- 7/3 is 2.333. This would mean that as X increases one unit, the likelihood of Y being 1 is multiplied by 2.333.
- Odds is different from probability. The probability of 1 arising in a 7/3 odds situation is 0.7 (7 / 7+3); the odds is 2.333.



# Poisson Regression

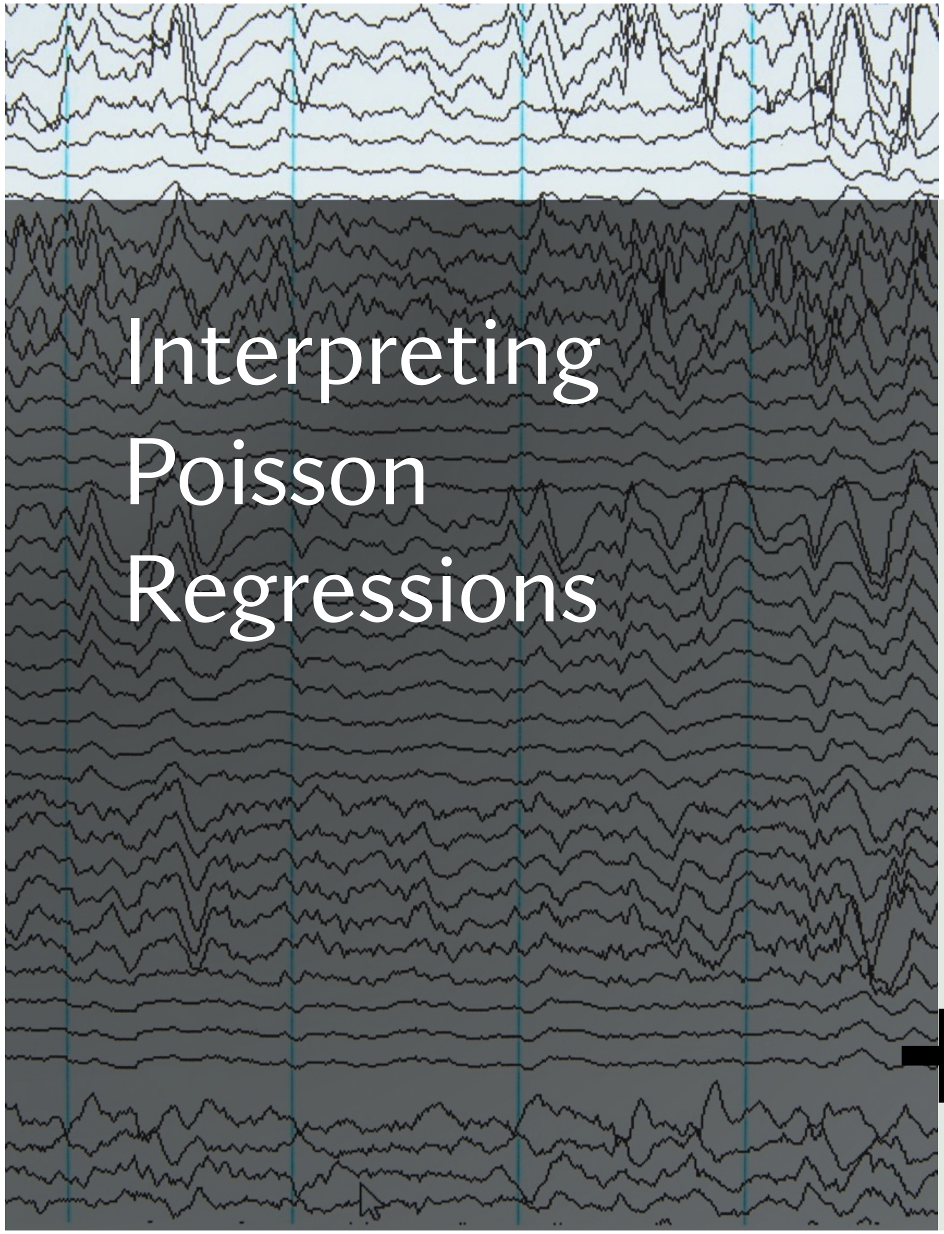
# Poisson Distribution

- The Poisson Distribution describes the likelihood of an event happening a certain number of times – for example, the number of buy or sell orders made on a stock exchange each second, or the number of daily deaths at a hospital.
- The shape of the distribution changes; when the mean is low, it is heavily skewed to the left (low values are more likely, with some high outliers).
- At higher mean values, the distribution becomes wider (meaning uncertainty is higher) and looks more ‘normal’.



## Counting Events

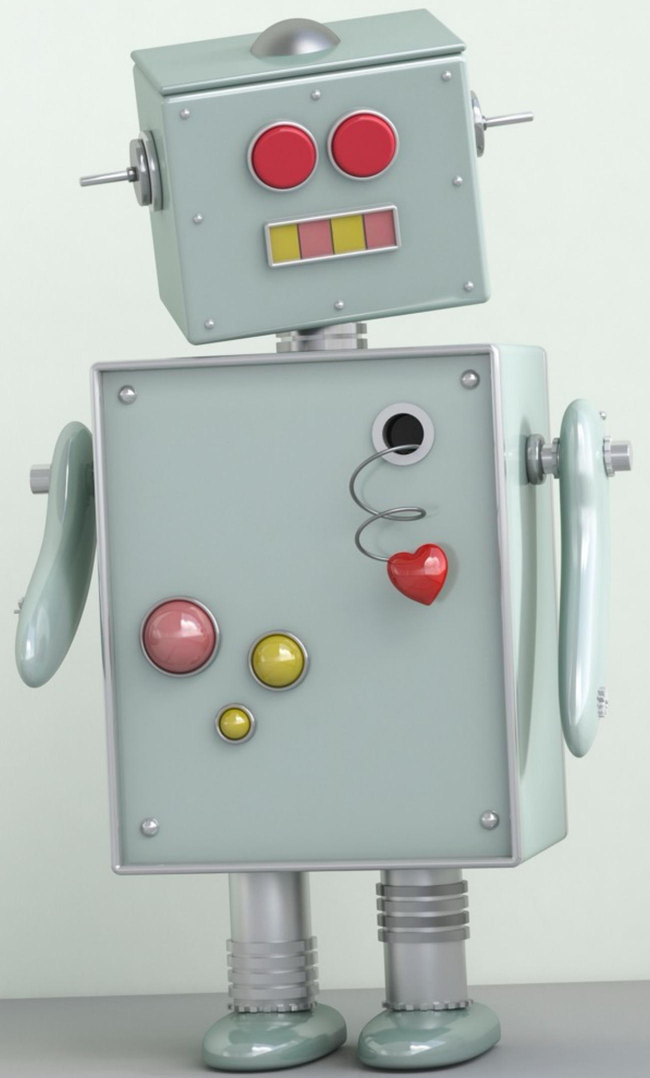
- When we want to model an outcome like this (counting how many times an event happens), we shouldn’t use standard OLS, because we don’t expect the outcome to be normally distributed.
- Instead in these cases we use Poisson Regression, which – much like logistic regression – is a variation on the regression system that assumes differently distributed data.



## Interpreting Poisson Regressions

- Poisson Regression results are very similar to logistic regression results, in that they provide coefficients in **log odds**.
- Therefore, similarly, you can use the exponent function to calculate the basic **odds** – and see how much we can expect the number of events to change for every unit increase in X.

## Over to RStudio



## Research Diaries

- Just a reminder! You should already be writing your research diaries / lab notes – giving your account of the progress of your team's project and explaining each major decision you took.
- You are free to include graphs etc. in these diaries – this is a good place to add graphs you made to understand the data but which you don't use in the final presentation.
- You should also include references to papers you read, etc.
- Don't be afraid to write about mistakes or mis-steps in the process!

## Research Updates

- You have a group milestone report due on Sunday night; I will get feedback to you as soon as possible, so you can report back again before the winter break.
- These assignments are not graded, but they're your opportunity to get direct feedback on the work you're doing and ensure that your team is moving in the right direction.
- These last two milestones are especially important, since you want to be sure the project work is proceeding properly over the winter break – in January we'll have minimal time to fix any issues that come up.