Introduction to Linear Regression

Signal Data Science

In this assignment, you'll learn run to do a linear regression, the simplest of all machine learning techniques!

UN Infant Mortality Dataset

We'll begin by using data from the United Nations about infant mortality in different countries for this analysis.

Refer to the code template in `day1Example.R` as you work through this assignment. You'll go through the existing code, alternating between figuring out what it does, answering questions, and writing your own code to supplement what's already there.

Getting started

First, we have some preliminaries to take care of.

- Install the packages `car`, `Ecdat`, `HistData`, `ggplot2`, `dplyr`, `Rmisc`, and `GGally`. Load `day1Example.R` in RStudio.
- Load the packages `car`, `ggplot2`, and `GGally`. Set `df = UN`.
- Print out the data contained in `df` by just typing `df` into the console. Run `View(df)` (case-sensitive!) to bring up a nicer GUI for viewing the dataset. Does anything stand out to you? If there are any questions you'd like to answer with this data, write them down as comments in your R file.
- R packages are extensively documented online. Look at the reference manual for the `car` package to read about the `UN` data.

Viewing correlations and cleaning the data

The first step in any data analysis is to clean the data! Data is often messy and incomplete, which gets in the way of doing any analysis. After data cleaning, it's

typically a good idea to look at some basic summary statistics before jumping directly into predictive modeling.

- Use `cor()` to find the correlation between infant mortality and GDP. There will be an error – what's wrong, and why is this happening? Look in the documentation for `cor()` and figure out how to tell it to ignore entries with missing values.
- For readability, multiply the correlation matrix by 100 and `round()` it to whole numbers.
- Wrap the above code into a function, `cor2()`, which outputs a correlation matrix (ignoring entries with missing values) with rounded whole numbers.

Instead of making each function we use handle missing values (`NA`s) in its own way, we can just remove all the rows in a dataset which contain missing values. If there are many such rows, this will decrease our sample size dramatically and potentially skew the results as well. However, if the proportion of rows with missing values is low, the effect of doing so will be minor.

- Type `?na.fail` and read the documentation on `NA`-related functions; find one appropriate for the job and use it to make `df2`, a data frame identical to `df` except without rows which contain missing values.

Visualizing distributions

Next, we'll do some visualizations and transformations of the data as a prelude to linear modeling.

- Use `ggpairs()` (from the `GGally` package) on the data frame.
  - What do you notice about the distributions of GDP and infant mortality?
  - Use `log()` on `df` to take the natural logarithm of every value in the data frame; assign the resulting data frame to `ldf`. Examine `ldf` with `ggpairs()`.
  - Note the differences and reflect on the appropriateness of a linear model for the untransformed vs. transformed data.

Running linear regressions

If you need a refresher on the theory of linear regression, briefly skim the relevant sections in *Applied Predictive Modeling*.

- Run the lines in the example code which use `lm()` to generate linear fits of infant mortality against GDP. You can type `linear_fit` and `summary(linear_fit)` in the console to get summaries of the results.

Note that `summary()` will print out a statistic denoted **Adjusted R-squared**, which can be interpreted as the *proportion of variance in the target variable explained by the predictors*.

- Briefly skim StackExchange to see how Adjusted R-squared is calculated.

In general, a higher Adjusted R-squared is better, because it means that our regression model captures more of the variance in the target variable.

Plotting linear regressions

To plot the results of a simple linear regression, it's actually easier to let `ggplot2` fit the model for you.

- Run the line starting with `ggplot...` to plot a scatterplot of the data in `df2` along with a linear fit of infant mortality to GDP.
  - What happens when you remove the `method` argument in `geom_smooth()`?
  - Look at the documentation for `geom_smooth()` and determine what method it defaults to.
  - Find the documentation online for that method, **briefly** read about it, and explicitly call it in the `method` argument instead of `"lm"`. How good of an approximation is a linear model?
  - Make the same plots for the linear fit of log(infant mortality) vs. log(GDP). (*Hint:* To access column `c` of a dataframe `df`, use the syntax `df$c`.)

Looking at the residuals

A residual is a fancy word for prediction error; it's the difference given by `actual - predicted`. After we fit a model to our data, visualizing the residuals allows us to easily check for evidence of heteroskedasticity.

- Run the first `qplot()` command, which plots the residuals (`actual - predicted` values) of the simple linear fit. Is there evidence of heteroskedasticity?
- Run the second `qplot()` command, which plots the residuals of the linear fit of the log-transformed data. Is the log-log transformation an improvement? Why or why not?

One of the assumptions of linear regression is that the variances of the distributions from which the errors are drawn have the same variance. If that's the case, then we shouldn't see much structure in the plot of the residuals. As such, if the residual plot looks very different from random noise, that's a warning sign

indicating that the linear model may not be working very well. For example, compare the top and bottom plots here (top has structure, bottom doesn't).

- Using the documentation and experimenting in the console, make sure you understand what `df$infant.mortality - exp(fitted(loglog_fit))` does.
- Generate and analyze a plot of residuals for the linear fit of log(infant mortality) vs. GDP.

Anscombe's quartet

Before moving on to another real-world dataset, we'll pause to briefly look at *Anscombe's quartet*, a famous dataset constructed in 1973 by Francis Anscombe.

- Use `read.csv()` to load `anscombe.csv`, in the `anscombe` dataset, into a variable called `df`. Again, type `df` into the console to see the data and the column names; column `y2` can be accessed with `df$y2`, etc.
- Compute the `mean()` and variance of each of the `x` and `y` columns.
- Compute the correlation of each of the four pairs of `x` and `y` columns.
- Compute the parameters of a linear regression of each `y` column against its corresponding `x` column – `y4 ~ x4`, `y3 ~ x3`, and so on and so forth.
- Use `plot()` to plot each of the four pairs of `x` and `y` columns – `y4` against `x4`, etc. Do the results surprise you?

Visualization is important! Summary statistics don't tell the whole story.

Galton's height data

Sir Francis Galton was a statistician who collected a large dataset on the heights of family members in the late 1800s. With this dataset, he was able to discover the phenomenon of regression to the mean.

With the Galton height dataset, you'll be getting more experience with basic operations on data frames and linear regressions.

Getting started

- Load the `HistData`, `dplyr`, and `Rmisc` packages and set `df = GaltonFamilies`. Take a look at it visually with `View(df)`.
- What variables does `df` include? Check the documentation to figure out what `midparentHeight` represents.

This time, the data frame has a lot of different columns. You can use `names()` to show the column names of `df` (you'll note that the output of `names(df)` is the same as the output of `colnames(df)`), and for a specific column `col` you can access it with the `$` operator, like so: `df$childHeight`. You can access and modify columns just like any other variable (with some small exceptions).

- Go back to the linear fits you made for the infant mortality dataset. Call `names(summary(linear_fit))` and figure out how to access the adjusted R-squared statistic directly instead of having to print out the entire summary of a linear fit every single time.

The `gender` variable is encoded as a **factor**, which we'll cover in greater depth later. For now, since we want to run linear regressions including gender, we want to turn it into a *binary numeric variable*, with values 0 and 1.

- Use a combination of arithmetic and `as.numeric()` to turn the `gender` variable into a column of 0s and 1s. Be sure to keep track of which gender you assign to each of 0 and 1.

Learning to use `dplyr`

The `dplyr` package is one of the most commonly used R packages and is particularly useful for the straightforward manipulation of data frames. The following tutorials (arranged in descending order of succinctness) should prove helpful:

- `tidyverse`'s own `dplyr` introduction.
- This tutorial, originally written for a biomedical data science class.
- CRAN's `dplyr` vignette.
- Hadley Wickham's `dplyr` tutorial. If you choose to make use of this tutorial, Pages 1 through 43 are most useful. Don't work through all the examples – just skim and refer back to the tutorial later if you need to.

Running linear regressions

In the questions below, "run a regression of A against X, Y and Z" should be understood as "first run a regression of A against X, Y and Z individually, perhaps in pairs, then run the regression against all three variables".

- Aggregate the data by family using the appropriate `dplyr` function. Before writing any R code to do so, think about what this aggregation actually means and what kinds of variables you want to calculate on a per-family basis.

  - The `Rmisc` package loads the `plyr` package as a dependency, which loads its own `summarise()` function and consequently overrides `dplyr`'s `summarize()`. To access the `summarize()` of dplyr, call `dplyr::summarise()`.

- Plot the average heights of children in each family against the mothers' heights, the fathers' heights, and the mid-parent heights. Afterward, use multiplot to display all three graphs at the same time.
  - The linked example uses `geom_line()`; ignore that part and just use it to learn about how to use `multiplot()` to combine plots.

- Compute and visually inspect the correlations between the variables in your aggregated dataset. Make a couple hypotheses about the data that you think you might be able to prove or disprove as you do further analysis and exploration.

- Compare the results of regressing average child heights against (fathers' heights and mothers' heights versus regressing against mid-parent heights. Interpret the results, paying particular attention to the adjusted R-squared statistic.

- Run a linear regression of the numbers of children in families against fathers' heights, mothers' heights, and average child heights. Looking at the summaries of the linear fits, do these regressions capture any statistically significant relationships? If so, with what p-values?
  - Here, "p-values" refers to the p-values associated with each coefficient in the regression, associated with testing for the null hypothesis that the value of each coefficient is equal to zero.

- Following the example usages of `ggplot()` that you've already seen along with the official documentation, use `ggplot()` in conjunction with `geom_histogram()` to make a histogram displaying the distribution of number of children per family.

Doing additional analysis

Let's return the original, *unaggregated* data. When we instruct you to run regressions, you should automatically assume that you should try to do as much interpretation as possible.

- Run a regression of child heights against mothers' heights, fathers' heights, mid-parent heights, and gender.

- Use the appropriate functions in `dplyr` to restrict to children of either gender and run regressions of child heights against mothers' heights and fathers' heights.

- Is it possible for us to analyze the effect of being firstborn vs. secondborn vs. thirdborn vs. ... on child heights? Why or why not?